Università degli Studi di Pisa

Facoltà di Scienze Matematiche Fisiche e Naturali
Corso di Laurea Specialistica in Informatica

Master Degree Thesis

Development of a Stochastic Simulator for Biological Systems Based on the Calculus of Looping Sequences

Candidate
Guido Scatena

Advisor
Dr. Antonio Cisternino

Advisor
Dr. Paolo Milazzo

Opponent
Dr. Laura Semini

Academic Year 2006/2007

" *The story so far :*
In the beginning the Universe was created.
This has made a lot of people very angry and been widely regarded as a bad move..."

The Restaurant at the End of the Universe
Douglas Adams

Abstract

Molecular Biology produces a huge amount of data concerning the behavior of the single constituents of living organisms. Nevertheless, this reductionism view is not sufficient to gain a deep comprehension of how such components interact together at the system level, generating the set of complex behavior we observe in nature. This is the main motivation of the rising of one of the most interesting and recent applications of computer science: *Computational Systems Biology*, a new science integrating experimental activity and mathematical modeling in order to study the organization principles and the dynamic behavior of biological systems.

Among the formalisms that either have been applied to or have been inspired by biological systems there are automata based models, rewrite systems, and process calculi.

Here we consider a formalism based on term rewriting called *Calculus of Looping Sequences* (CLS) aimed to model chemical and biological systems. In order to quantitatively simulate biological systems a stochastic extension of CLS has been developed; it allows to express rule schemata with the simplicity of notation of term rewriting and has some semantic means which are common in process calculi.

In this thesis we carry out the study of the implementation of a stochastic simulator for the CLS formalism. We propose an extension of Gillespie's stochastic simulation algorithm that handles rule schemata with rate functions, and we present an efficient bottom–up, pre–processing based, CLS pattern matching algorithm.

A simulator implementing the ideas introduced in this thesis, has been developed in F#, a multi–paradigm programming language for .NET framework modeled on OCaml. Although F# is a research project, still under continuous development, it has a product quality performance. It merges seamlessly the object oriented, the functional and the imperative programming paradigms, allowing to exploit the performance, the portability and the tools of .NET framework.

Contents

List of Figures

Index of Listings

Introduction

0.1 Definitions and Motivations

Molecular Biology studies produce a huge amount of data concerning the behavior of the single constituents of living organisms. With the help of *Bio Informatics*, organizing and analyzing such data, we are assisting to an increase of the understanding of the functional behavior of such constituents. Nevertheless, this is not sufficient to gain a deep comprehension of how such components interact together at the system level, generating the set of complex behavior we observe in nature. In fact the new international scientific panorama is characterized by the convergence of many disciplines helped by Computer Science like qualifying factor. The change of the research paradigm in Biology from the reductionist approach to the systemic one imposes a similar adaptation also to the Computer Science technologies of reference. This is the main motivation of the rising of one of the most interesting and recent applications of computer science: the *Computational Systems Biology*, a new science integrating experimental activity and mathematical modeling in order to study the organization principles and the dynamic behavior of biological systems (see [58] for an introduction to the field).

Mathematical and computational techniques are central in this approach to Biology, as they provide the capability of describing formally living systems and studying their properties. Clearly, a correct choice of the abstraction level of such description is fundamental in order to grasp the key principles without getting lost in the details (see [98, 57, 23] for an introduction to the art of modeling biological systems).

The great challenge of system biology is to understand whether models, originally developed for describing systems of interacting components, can be applied for modeling and analyzing biological systems. This could offer to biologist very useful simulation and verification tools that could replace expensive experiments *in vitro* or guide the experiments by making prediction on their possible results. Such

tools could help biologist to better understand of both qualitative and quantitative evolution of complex biological systems, such as pathways and networks of proteins and this is also essential for testing the effect of medicines or enzymes and for characterize regulators factors (see [30] for example). Biologists could make *in silico*[1] experiments that has an edge over conventional experimental Biology in terms of cost, ease and speed; he could also experiments the interaction between new artificial element and known disease causes. Moreover, unknow constants can be found using these instruments as tools for reverse engineering.

In computational systems biology many different modeling techniques are used in order to capture the intrinsic dynamics of biological systems. These modeling techniques are very different, both in the spirit and in the mathematics they use. Some of them are based on the well known instrument of *differential equations* and therefore they represent phenomena as continuous and deterministic (see [49, 41] for a survey). On the other side we find stochastic and discrete models, that are usually simulated with Gillespies algorithm [44], tailored for simulating (exactly) chemical reactions. In the middle, we find hybrid approaches like the Chemical Langevin Equation (see [45]), a stochastic differential equation that bridges partially these two opposite formalisms.

The parallel between Biology and Computer Science can be seen quite easy: the biology is the study of biological systems, complex interactive systems regarding living organisms, can take advantage of tools and methods created by Computer Science, that studies complex interactive systems. For example genetic networks can be seen as calculation networks in which the computation it is forwarded by proteins, produced through an elaborated process of genetic expression (see Section 1.1.3). Thus biology can take advanced of languages developed for describing complex parallel computer systems for deriving languages describing complex parallel biological systems.

[1] Performed on computer or via computer simulation

Like Philips and Cardelli says [61, 68]:

> *"in many ways, biological systems are like massively parallel, highly complex, error-prone computer systems".*

In summary the goals of biological simulation by Computational System Biology are :

- generate knowledge through simulations to discover not known or not measurable *in vitro* parameters (*reverse engineering* approach);
- allow the development of new artificial elements to make interact with the known disease causes in order to alter negatives behaviors.

The main difficulties are instead :

- corrected formulation of the biologic dynamics, which are for their nature subordinates to molecular noise;
- efficient algorithms, computational equipped for the complexity of the dynamics of the considered systems.

Among the formalisms that either have been applied to or have been inspired by biological systems there are automata-based models [13, 73], rewrite systems [32, 86], and process calculi [85, 87, 28, 23].

A new formalism, called *Calculus of Looping Sequences* (CLS for short), for describing biological systems and their evolution, has been developed [16, 77, 17, 19, 21]. CLS is based on term rewriting and has some features, such as a commutative parallel composition operator, and some semantic means, such as bisimulations [20], which are common in process calculi. This permits to combine the simplicity of notation of rewriting systems with the advantage of a form of compositionality. Moreover, the CLS formalism is extended with a stochastic semantics in the *Stochastic Calculus of Looping Sequence* (SCLS).

Cellular pathways mainly consist of protein–protein interaction, where proteins can be located either inside, outside or on the surface of a membrane. The Stochastic Calculus of Looping Sequence, allowing dealing directly with proteins wherever they are located, is suitable to describe microbiological systems and their evolution. Systems are represented by terms. The evolution of a term is made by a set of rewrite rules enriched with stochastic rates representing the speed of the activities described by the rules, and can be simulated automatically.

0.2 Contributions

We present the study of the implementation of a simulator for the stochastic version of CLS; a formalisms for describing biological systems, that allows to express *rule schemata*, that are rules with typed variables and rate functions which value depends on the instantiation of variables.

We have extended the Gillespie's stochastic simulation algorithm to take account of rule schemata and we have developed an efficient pattern matching algorithm. This algorithm is based on the bottom up approach proposed in the seminar paper of Hoffmann and O'Donnel [50] and has some important characteristics for efficiency like good response to local changes in the simulation state.

We have developed a simulator in which the ideas proposed in this thesis has been implemented.

0.3 Related Works

All the work of this thesis start from the papers by Barbuti et al. [18, 19, 77].

This is framed in a wider picture in which we find the intense activities of many research groups about systems biology. In this within many formalisms have been developed, that either have been applied to or have been inspired by biological systems.

A wide range of software tools aim to carry out simulations exists, among which we can cite as an example [81].

0.4 Published Software

To show the ideas proposed in this thesis we have developed a software simulator, called SCLSm (Stochastic CLS Machine), that can be found on
`http://www.di.unipi.it/~milazzo/biosims/`

0.5 Structure of the Thesis

The thesis is structured as follows

- In Chapter 1 we give some notions of Biology and System Biology (1.1), of probabilistic theory (1.2), of combinatorics (1.4) and of the simulation of biological system (1.3), that will be assumed in the rest of the thesis.

- In Chapter 2 we introduce the *Calculus of Looping Sequences*, a formalism based on term rewriting suitable for describing qualitative aspects of biological and biochemical systems.

- In Chapter 3 we present all the issues we have found in the development and implementation of a stochastic simulator for CLS. We present an efficient pattern matching algorithm for CLS pattern matching and an extension of the Gillespie's stochastic simulation algorithm that deals with rule schemata.

- In Chapter 4 we look at some simulation results that prove that the developed simulator has a correct behavior, and in Chapter 5 show through benchmarks that the ideas proposed for the implementation are valid.

- Finally, we give some conclusions and discussions about further works in Chapter 6.

In Appendix A there are an overview of the software developed with the instruction to use.

Part I

BACKGROUND

Chapter 1

Background

In this chapter we briefly introduce some notions of Biology and System Biology (1.1), of probabilistic theory (1.2), of combinatorics (1.4) and about the simulation of biological system (1.3).

1.1 Introduction to Computational System Biology

1.1.1 Actors

The central unit of System Biology are cells. There are two basic classifications of cell: *procaryotic* and *eucaryotic* (see Figure:1.1). The distinguishing feature between the two types is that eucaryotic cell possesses a membrane-enclosed nucleus and a procaryotic cell, usually small and relatively simple, does not. Procaryotes are considered representative of the first types of cell to arise in biological evolution, and include for instance almost all bacteria. Eucaryotic cells, are generally larger and more complex, expressing an advanced evolution, and include fungi, algae and multicellular plants and animals. In eucaryotic cells, different biological functions are segregated in discrete regions withing a cell, often in membrane-limited structures. Sub cellular structures which have distinct organizational features are called *organelles*. In contrast, procaryotic cells have only a single cellular membrane and thus no membranous organelles. Moreover one molecular difference between the two types of cells is noticeable in their genetic material: procaryotes have a single chromosome, while eucaryotes possess more than one chromosome.

Excluding small molecules, like water and minerals, there are essentially the following classes of molecules that interacts inside cells.

Nucleic Acids : DNA and RNA. *deoxribonucleic acid* (DNA) and *ribonucleid acid* (RNA) are polymeric chains (see Figure 1.2), more precisely they are

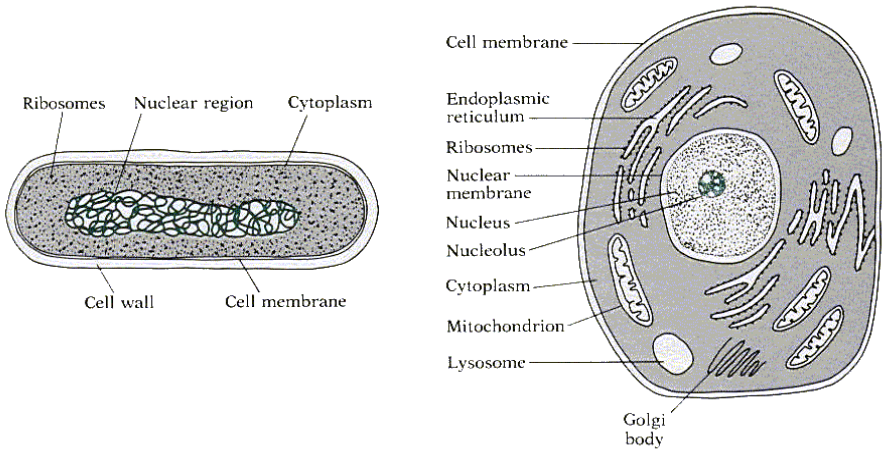


Figure 1.1: Procaryotic and eucaryotic cells; are indicated some examples of *organelles* .

chains of five *nucleotides*: *adenine*, *cytosine*, *guanine*, *thymine*, and *uracil*. The nucleotides are simple organic molecules constituted from a sugar, a phosphate group, and a nitrogenous base. DNA stores the genetic information, while RNA participates in the process of extraction of this information, being the bridge molecule in the mechanism of construction of proteins from DNA. In eucaryiotic the DNA is placed in the nucleus and is shaped as double helix, while in procaryotic it is placed directly in the cytoplasm and it is circular.

Proteins. Proteins are sequences made up from about twenty different *amino acids*, that fold up in peculiar three dimensional shape (see Figure 1.3) that is determined according some energetically stability; this shape is most often the conformation requiring the least amount of energy to maintain and can be more or less stable. In this complex three dimensional shape is something possible to identify places where chemical interaction with other molecules can occur. This places are called *interaction sites*, and are usually the basic entities in the abstract description of the behavior of a protein. An eucaryotic or procaryotic cell contains thousand of different proteins; the genetic information contained in chromosomes determines the protein composition of an organism. The best-known role of proteins in the cell is their duty as enzymes, which catalyze chemical reactions, even if some of these have function of transport, storage and cellular structure.

Enzymes. Are proteins that behave as very effective catalyst, and are responsible

for thousand of coordinated chemical reactions involved in biological processes of living systems. Like any catalyst, an enzyme accelerates the rate of reaction by lowering the energy of activation required for the reaction to occur (see Figure 1.4). Moreover, as a catalyst, an enzyme is not destroyed in the reaction and therefore remains unchanged and is reusable after the reaction is occurred. Enzymes are usually highly specific catalysts that accelerate only one or a few chemical reactions. Enzymes effect most of the reactions involved in metabolism and catabolism as well as DNA replication, DNA repair, and RNA synthesis. Also in the external membrane of a cell there are enzymes responsible for transporting some molecules from the outside to the inside and vice-versa.

Lipids. Lipids have mainly a structural function. In particular, cellular membranes are constituted by a double layer of phospholipids (a class of lipids), whose structure gives to membranes unique properties, like the ability to self-assemble, to merge one into the other, and to contain interface proteins in their surface.

Carbohydrates. These are sugars, involved mainly in the energetic cycle of the cell. They are often linked into complex structures, some of them having with tree-like shapes.

These classes of molecules can be seen as the basic of some different abstract machine [27]. A common feature of these abstract machines is that the interactions among the substances involved form complex networks, whose structure is one of the main responsible of the cellular dynamics. These networks show an highly non-linear behavior, induced by the presence of several feedback and feed-forward loops. Another property of these networks is redundancy; for instance, many genes can encode the same protein, though they are expressed in different conditions. Moreover important characterizing features are modularity and structural stability. All these properties are responsible, for instance, of the amazing robustness exhibited by living beings in resisting and adapting to perturbations of the environment [58].

1.1.2 Biochemical reaction networks

Biochemical reactions are chemical reactions involving mainly proteins. As in a cell there are several thousands of different proteins, the number of different reactions that can happen concurrently is very high. Depicting a diagram of such interaction

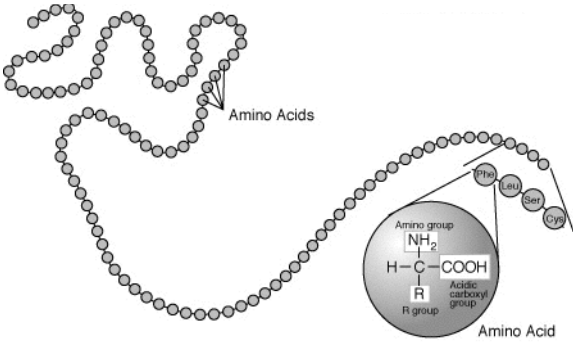


Figure 1.2: A polymeric chain is formed by amino acids.

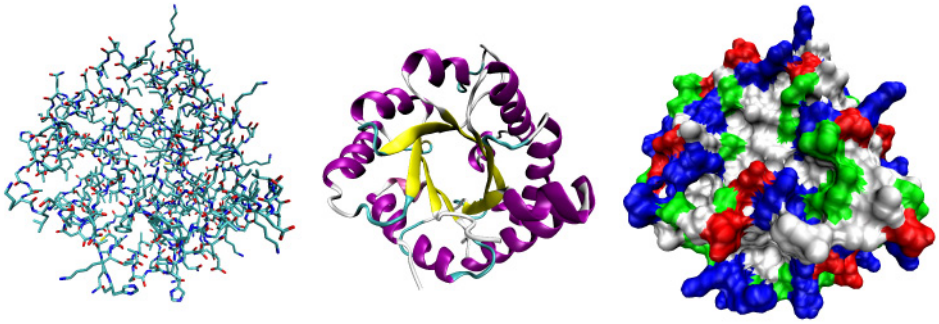


Figure 1.3: Schematic views of the three dimensional structure of protein.

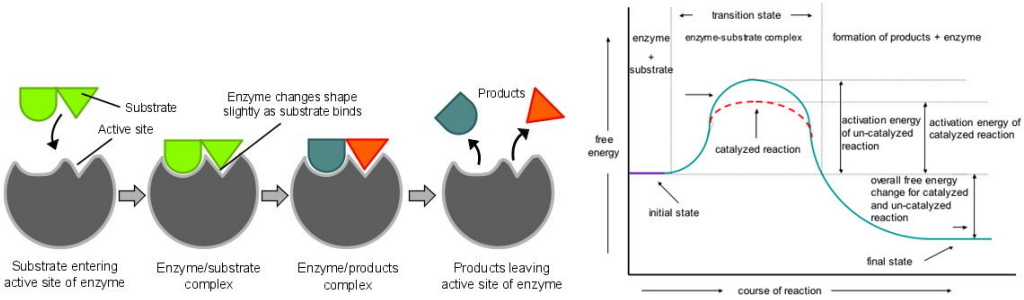


Figure 1.4: Catalyzed reaction by effect of an enzyme. On the left we can see that the enzyme acts like a catalyst doing like placeholder for reactants and remaining unchanged after the reaction has occurred. On the right we can see how the presence of enzyme reduce the activation energy of the reaction.

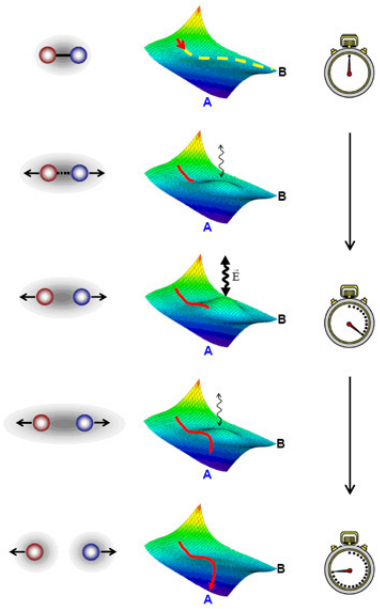


Figure 1.5: Dynamic of energy in a chemical reaction; we can see how the evolution of the reaction always follows dynamically the way that requires less energy.

capabilities, we obtain a complex network having all the characteristics mentioned at the end of the previous section.

The fundamental empirical law governing reaction rates in biochemistry is the *law of mass action*. This states that for a reaction in a homogeneous medium, the reaction rate will be proportional to the concentrations of the individual reactants involved. A chemical reaction is usually represented by the following notation:

$$\ell_1 R_1 + \ldots + \ell_k R_k \underset{k_{-1}}{\overset{k}{\rightleftharpoons}} \ell_{p1} P_1 + \ldots + \ell_{pk} P_k$$

where $R_1, \ldots, R_k, P_1, \ldots, P_k$, are molecules, $\ell_1, \ldots, \ell_k, \ell_{p1}, \ldots, \ell_{pk}$ are their stoichiometric coefficients; R_i are the reactants, P_i are the products and k and k_{-1} are the kinetic constants that are related to the kinetic model adopted and representing its basic expected frequency. The equation above states that combining an appropriate number of reactants we can obtain the products. The use of the symbol $\rightleftharpoons$ denotes that the reaction is *reversible* (i.e. it can occur in both directions). Irreversible reactions are denoted by the single arrow $\rightarrow$.

Example 1.1. Given the simple reaction

$$2A \underset{k_{-1}}{\overset{k}{\rightleftharpoons}} B$$

the rate of the production of molecule B for the law of mass action is:

$$\frac{dB_+}{dt} = k[A]^2$$

and the rate of destruction of B is:

$$\frac{dB_-}{dt} = k_{-1}[B]$$

where $[A], [B]$ are the *concentrations* (i.e. moles over volume unit) of the respective molecules.

In general, the rate of a reaction is:

$$k[S_1]^{\ell_1} \cdots [S_\rho]^{\ell_\rho}$$

where $S_1, \ldots, S_\rho$ are all the distinct molecular reactants of the reaction.

The rate of a reaction is usually expressed in $moles \cdot s^{-1}$ (it is a speed), therefore the measure unit of the kinetic constant is $moles^{-(L-1)} \cdot s^{-1}$, where we denote with L the sum of the stoichiometric coefficients, that is the total number of reactant molecules.

The interactions between elements always follows the way that needs less energy to happen (see Figure1.5). In fact to make a reaction occur it is necessary to overcome its *activation energy*. It is necessary that the collision between two or more molecules happens; they must be opportunely oriented, endowed of a minimum of energy such to exceed the repulsive electric forces generated from their electron cloud. This quantity of energy is the *activation energy* of the reaction. If the energy is enough, the involved molecules (*reactants*) can get so close each other to be able to reorganize their own bounds into new compounds (*products* of the reaction).

The activation energy allows the molecules that collide to form the so–called *activated complex* or *state of transition*, whose existence is extremely short ($\sim 10^{-15}$ seconds); from this state it is possible that the original bounds are reformed or that products are originated. Also exists multi-state reactions in which the passage from reactants to products implies the formation of more transition states; in these cases the activation energy required is the highest of each intermediate transition state activation energy.

In case of reversible reactions, in which the speed of the inverse reactions are similar, can happen that the system reaches a *dynamic equilibrium*. In case of dynamic equilibrium the system is not static (there are continuously reactions and

$$S_1 + E_1 \rightarrow S_2 + E_1$$
$$S_2 + E_2 \rightarrow S_1 + E_2$$

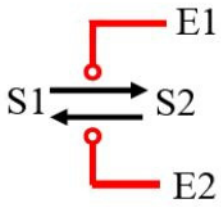


Figure 1.6: Example of reversible reaction expressed by Kohn's formalism.

inverse reactions) but the properties of the system, the amount of the molecular populations, do not change.

A different way to represent biochemical reactions uses a graphical notation. One of the most renown is the one developed by Kohn [63, 64], using different kind of arrows and having an implicit representation of complexes (see Figure 1.6), or by Kitano in [59]; see [65] for a comparison of these two formalism.

1.1.3 Genetic Regulatory Networks

We had seen that DNA contains instructions for the biological development of a cellular form of life, RNA carries information from DNA to sites of protein synthesis in the cell and provides amino acids for the development of new proteins, proteins perform activities of several kinds in the cell. Schematically we have this flux of information:

$$\text{DNA} \xrightarrow{\text{transcription}} \text{RNA} \xrightarrow{\text{translation}} \text{Protein}$$

in which *transcription* and *translation* are the activities of performing a "copy" of a portion of DNA into a mRNA (RNA messenger) molecule, and of building a new protein by following the information found on the mRNA and by using the amino acids provided by tRNA molecules (RNA transfer). This process is known as the *Central Dogma of Molecular Biology.*

Even if the central dogma in molecular biology states that DNA is transcribed into RNA, which is then translated into proteins, the flow of information, however, goes also in the other direction: proteins interact with the DNA transcription mechanism in order to regulate it. DNA is composed by several (thousand of) genes, each encoding a different protein. Genes are roughly composed by two regions: the *regulatory region* and the coding one. The *coding region* stores the information needed to build the coded protein using the genetic code to associate triples of nucleotides

to amino acids. The regulatory region, usually found upstream of the coding part of the gene and called *promoter*, is involved in the mechanisms that control the expression of the gene itself. The regulation is performed by dedicated proteins, called *transcription factors*, that bind to specific small sequences of the promoter, called *transcription factors binding sites*. There are basically two different kinds of transcription factors, the *enhancers*, increasing or enabling the expression of the coded protein, and the *repressors*, having a negative control function. Transcription factors act on DNA in complex ways, often combining in complexes before the binding or during it. Also binding sites, like interaction sites on proteins, can be exposed or covered, both by the effect of a bound factor or by conformational properties of the DNA strand, though this last form of regulation is not well understood yet.

1.2 Notions of Probability Theory

A *probability distribution* is a function which assigns to every interval of the real numbers a probability $P(I)$, so that Kolmogorov axioms are satisfied, namely:

- for any interval I it holds $P(I) \geq 0$
- $P(\mathbb{R}) = 1$
- for any set of pairwise disjoint intervals $I_1, I_2, \ldots$ it holds $P(I_1 \cup I_2 \cup \ldots) = \sum P(I_i)$

A random variable on a real domain is a variable whose value is randomly determined. Every random variable gives rise to a probability distribution, and this distribution contains most of the important information about the variable. If X is a random variable, the corresponding probability distribution assigns to the interval $[a, b]$ the probability $P(a \leq X \leq b)$, i.e. the probability that the variable X will take a value in the interval $[a, b]$. The probability distribution of the variable X can be uniquely described by its *cumulative distribution function* $F(x)$, which is defined by

$$F(x) = P(X \leq x)$$

for any $x \in \mathbb{R}$.

A distribution is called *discrete* if its cumulative distribution function consists of a sequence of finite jumps, which means that it belongs to a discrete random variable X: a variable which can only attain values from a certain finite or countable set.

A distribution is called *continuous* if its cumulative distribution function is continuous, which means that it belongs to a random variable X for which $P(X = x) = 0$ for all $x \in R$.

Most of the continuous distribution functions can be expressed by a probability density function: a non-negative Lebesgue integrable function f defined on the real numbers such that

$$P(a \leq X \leq b) = \int_a^b f(x)\,dx$$

for all a and b.

The *support* of a distribution is the smallest closed set whose complement has probability zero.

An important continuous probability distribution function is the *exponential distribution*, which is often used to model the time between independent events that happen at a constant average rate. The distribution is supported on the interval $[0, \infty)$. The probability density function of an exponential distribution has the form

$$f(x, \lambda) = \begin{cases} \lambda e^{-\lambda x} & x \geq 0 \\ 0 & x < 0 \end{cases}$$

where $\lambda > 0$ is a parameter of the distribution, often called the rate parameter.

The cumulative distribution function, instead, is given by

$$F(x, \lambda) = \begin{cases} 1 - e^{-\lambda x} & x \geq 0 \\ 0 & x < 0 \end{cases}$$

The exponential distribution is used to model Poisson processes, which are situations in which an object initially in state A can change to state B with constant probability per unit time λ. The time at which the state actually changes is described by an exponential random variable with parameter λ. Therefore, the integral from 0 to T over f is the probability that the object is in state B at time T.

In real-world scenarios, the assumption of a constant rate (or probability per unit time) is rarely satisfied. For example, the rate of incoming phone calls differs according to the time of day. But if we focus on a time interval during which the rate is roughly constant, such as from 2 to 4 p.m. during work days, the exponential distribution can be used as a good approximate model for the time until the next phone call arrives.

The *mean* or *expected value* of an exponentially distributed random variable X with rate parameter λ is given by

$$\mathrm{E}[X] = \frac{1}{\lambda}$$

In light of the example given above, this makes sense: if you receive phone calls at an average rate of 2 per hour, then you can expect to wait half an hour for every call.

1.3 Simulation of Biological Systems

In general terms the problem of the simulation of a biological system can be formalized in the following way:

Definition 1.2 (***Biological System Simulation Problem***)**.** in a fixed volume V, containing a mixture uniform distributed of N species that can interact through M reaction channels, knowing the molecule number of every species to the time t_0, we want to know the number of molecules to the time t_1.

From [44] by Gillespie

> *"There are two formalisms for mathematically describing the time behavior of a spatially homogeneous chemical system: the deterministic approach regards the time evolution as a continuous, wholly predictable process which is governed by a set of coupled, ordinary differential equations (the reaction-rate equations); the stochastic approach regards the time evolution as a kind of random-walk process which is governed by a single differential-difference equation (the master equation). Fairly simple kinetic theory arguments show that the stochastic formulation of chemical kinetics has a firmer physical basis than the deterministic formulation, but unfortunately the stochastic master equation is often mathematically intractable. There is, however, a way to make exact numerical calculations within the framework of the stochastic formulation without having to deal with the master equation directly."*

Traditionally this kind of systems are studied by deterministic simulation systems based on differential equation; unfortunately the biologicals systems they are characterized by two type of noise: intrinsic noise due to the same nature of the reactions, extrinsic due to the conditions of the external atmosphere (see [22, 80, 9] and [75] for a survey on the evidence of the stochasticity nature of biological systems). This causes that in the chemical reactions in general, and in genetic regulatory networks (that are systems of enzymatic reactions) in particular, the knowledge of the

molecular mechanisms is not sufficient to predict with certainty the future of a cellular population through deterministic simulation. Like the launch of a dice: it is not possible to know with certainty the result; for practical purposes can be useful seen it as random phenomenon and to reason on probabilities of a stochastic system.

1.3.1 Deterministic Simulation

In the classical setting, the amount of a chemical substance is generally regarded as a concentration, thus expressed in units like moles per liter. This is probably a heritage of the experimental study of chemical reactions in beakers of water. Conventionally, the concentration of a chemical species X is denoted $[X]$. The law of mass action kinetics states that the rate of a chemical reaction is directly proportional to the product of the effective concentrations of each participating molecule. Basically, it is proportional to the concentration of reactants involved raised to the power of their stoichiometry. If we consider all the reactions involved, we can write a system of differential equations giving the speed of change of the concentration of each molecule. To write the equation for X, we simply have to sum the rate law terms of the different equations involving X, multiplying by $+1$ all the terms coming from equations where X is a product and by -1 the terms corresponding to equations where X is a reactant. Once we have this set of equations, we can analyze it. For instance, we can solve it numerically for given initial conditions, we can study its equilibrium points, or we can analyze its dependence on parameters, looking for bifurcation points and chaotic regions (see [92]).

1.3.2 Stochastic Simulation

Stochastic Simulation of Chemical Reactions by Gillespie's Algorithm

The Gillespie's *direct method* (also know as *stochastic simulation algorithm*, SSA in short), proposed in [44], is an algorithm that generates a trajectory in the state space (that is one possible solution) statistically correct of a stochastic equation (the *master equation*). The direct method is a variant of the *dynamic method of Monte Carlo* (more precisely of the *kinetic Monte Carlo* [29, 107]) that, contrarily to the traditional methods, that are continues and deterministic and capable only to simulate the interaction of million molecules, allows a discrete, stochastic simulation, of systems with little number of reactants subordinates to molecular trouble; in fact each reaction is explicitly simulated and the molecular trouble is modeled through the stochastic behavior.

In [44] Gillespie gives a stochastic formulation of chemical kinetics that is based on the theory of collisions and that assumes a stochastic reaction constant c_μ for each considered chemical reaction R_μ. The reaction constant c_μ is such that $c_\mu dt$ is the probability that a particular combination of reactant molecules of R_μ will reacts in an infinitesimal time interval dt.

The probability that a reaction R_μ will occur in the whole solution in the time interval dt is given by $c_\mu dt$ multiplied by the number of distinct R_μ molecular reactant combinations. For instance, the reaction

$$R_1 : S_1 + S_2 \rightarrow 2S_1 \tag{1.1}$$

will occur in a solution with X_1 molecules S_1 and X_2 molecules S_2 with probability $X_1 X_2 c_1 dt$. Instead, the inverse reaction

$$R_2 : 2S_1 \rightarrow S_1 + S_2 \tag{1.2}$$

will occur with probability $\frac{X_1(X_1-1)}{2!}c_2 dt$. The number of distinct R_μ molecular reactant combinations is denoted by Gillespie with h_μ, hence, the probability of R_μ to occur in dt (denoted with $a_\mu dt$) is

$$a_\mu dt = h_\mu c_\mu dt \ .$$

Now, assuming that $S_1, \ldots, S_n$ are the only molecules that may appear in a chemical solution, a *state* of the simulation is a tuple $(X_1, \ldots, X_n)$ representing a solution containing X_i molecules S_i for each i in $1, \ldots, n$. Given a state $(X_1, \ldots, X_n)$, a set of reactions $R_1, \ldots, R_M$, and a value t representing the current time, the algorithm of Gillespie performs two steps:

1. The time $t + \tau$ at which the next reaction will occur is randomly chosen with τ exponentially distributed with parameter $\sum_{\nu=1}^{M} a_\nu$;

2. The reaction R_μ that has to occur at time $t + \tau$ is randomly chosen with probability $a_\mu dt$.

The function $P_g(\tau, \mu)dt$ represents the probability that the next reaction will occur in the solution in the infinitesimal time interval $(t + \tau, t + \tau + dt)$ and will be R_μ. The two steps of the algorithm imply

$$P_g(\tau, \mu)dt = P_g^0(\tau) \cdot a_\mu dt$$

where $P_g^0(\tau)$ corresponds to the probability that no reaction occurs in the time interval $(t, t+\tau)$. Since $P_g^0(\tau)$ is defined as

$$P_g^0(\tau) = \exp\left(-\sum_{\nu=1}^{M} a_\nu \tau\right)$$

we have, for $0 \leq \tau < \infty$,

$$P_g(\tau, \mu)dt = \exp\left(-\sum_{\nu=1}^{M} a_\nu \tau\right) \cdot a_\mu dt \ .$$

Finally, the two steps of the algorithm can be implemented in accordance with $P_g(\tau, \mu)$ by choosing τ and μ as follows:

$$\tau = \left(\frac{1}{\sum_{\nu=1}^{M} a_\nu}\right) \ln\left(\frac{1}{r_1}\right) \qquad \mu = \text{the integer for which } \sum_{\nu=1}^{\mu-1} a_\nu < r_2 \sum_{\nu=1}^{M} a_\nu \leq \sum_{\nu=1}^{\mu} a_\nu$$

where $r_1, r_2 \in [0, 1]$ are two real values generated by a random number generator. After the execution of the two steps, the clock has to be updated to $t + \tau$ and the state has to be modified by subtracting the molecular reactants and adding the molecular products of R_μ.

See Listing 3.5 for the code of Gillespie algorithm.

Weak Points and Alternatives

Although the Gillespie's algorithm solves the Master Equation exactly, it requires substantial amount of computational effort to simulate a complex system. Three situations cause an increase of computational effort in the Gillespie's algorithm. These conditions decrease the step time of each iteration thus forcing the algorithm to run for larger number of iterations to simulate a given experiment. The conditions are as follows:

- increase in the number of possible reactions,
- increase in the number of molecules of the species,
- faster rate of the reactions.

These factors cause great disparities in the timescale of reaction channels in the system (see Paragraph 4.5.1 for a concrete example of this phenomenon). In fact, in order to maintain the exactness of the simulation, the size of τ proportionally decreases with respect to the arising of the complexity of the system. In other words, the algorithm requires shorter time steps to capture the fast dynamics of the system. Thus, the difference in the time–scales between different reaction channels may cause substantial computational difficulties.

Since the development of the *direct method* in [44], many variants of the stochastic algorithm have been proposed in order to improve performance (see Figure 1.7). For

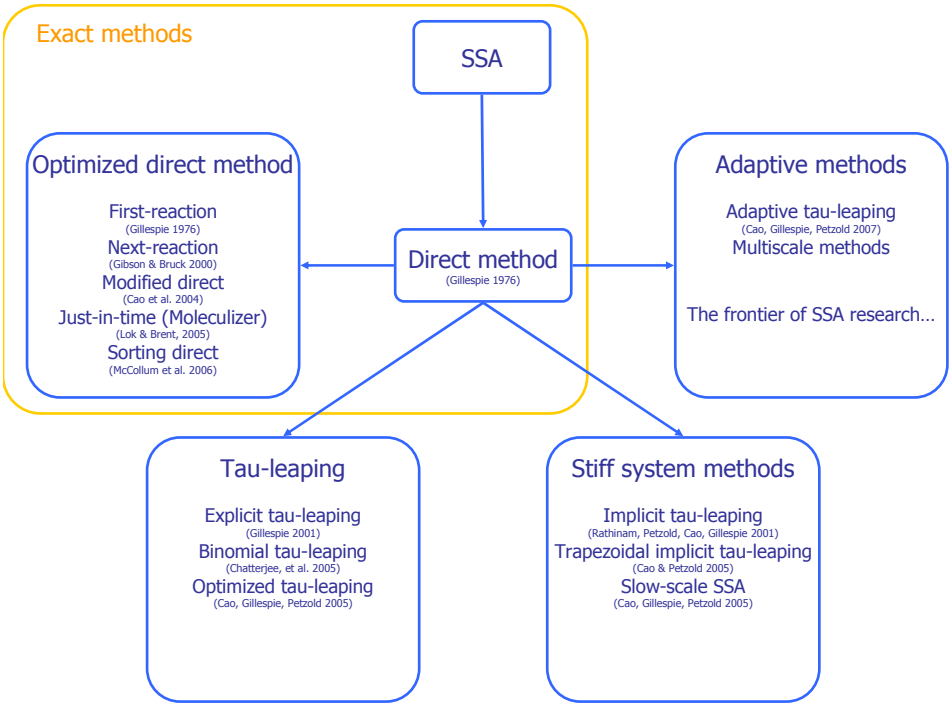


Figure 1.7: Flowchart of Stochastic Simulation Algorithms (from [84]).

example while direct method is an exact procedure, the *Tau-Leaping Method*, developed by Gillespie [46, 26], is an approximation taking larger time leaps. More than one reactions are executed in the time interval τ; the order in which they are executed is not important. Although the procedure for computing τ is more complicated and therefore more time consuming, with this method the larger leaps result in fewer steps which imply faster, but approximated, simulations. The critical point in this algorithm is taking sufficiently large τ values to improve efficiency while keeping it small enough that the propensity functions do not appreciably change in the interval.

1.3.3 Comparison of the Two Models of Simulation

The stochastic model of simulation is always valid when the deterministic one is, and is valid also when the deterministic one is not. Moreover it has the characteristic of being discrete in the time and in the amounts of the populations, and therefore being more realistic. Another characteristic that makes stochastic simulation more realistic is that it introduces variations between different runs; in contrast the deterministic simulation will always follow the same identical trajectory in the states space.

The main defect of stochastic model of simulation is to be computational expensive as it simulates every reaction explicitly. See the following example and Figure 1.8 for a comparison between the two models of simulation.

Example 1.3. By executing the same simulation by the two simulation methods we

can obtain very different behavior, especially when we consider populations made of a small number of molecules. We have the following set of evolution rules, also known as the Lotka-Volterra systems (see [44]).

$$A \xrightarrow{k_1} 2A$$

$$A + B \xrightarrow{k_2} 2B$$

$$B \xrightarrow{k_3} \emptyset$$

This system of equations aims to model a simple prey-predator dynamics, where A (wolf) is the prey and B (rabbit) is the predator.

Deterministic and stochastic simulations of these rules, with $k1 = 1$ and $k2 = k3 = 0.1$, starting from the same state with $10B$ and $4A$, give two very different results (see Figure 1.8). The main cause of this difference is that the deterministic simulation consider the amount of each reactants as a continuous value. This cause that a certain population is still present, and contribute to reactions, even if are present in number less than one, carrying to a not realistic behavior (0.6 rabbit will never produce 0.3 child !!)

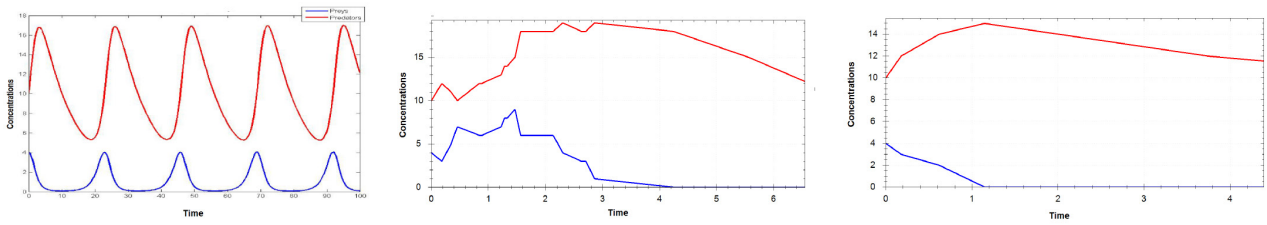


Figure 1.8: A comparison between the deterministic model and the stochastic one. On the left we can see the simulation made by deterministic simulator (from [23]) whereas the other two plot on the right are made by two runs of the developed stochastic simulator. We can see how in the deterministic simulator a population can exist even if have less than one member. We can also see how different runs of stochastic simulator can give different results although the deterministic simulator give always same behavior. The first graph it is obtained by a simulator based on differential equations, whereas the other two graphs are made by the stochastic simulator that we have developed.

In summary the two methodologies are compared in the following table.

	Deterministic Simulation	**Stochastic Simulation**
Time	continue	discrete
Space	continue	discrete
Application	correct for millions of molecules in a isolated environment	always correct
Complexity	low	high (each possible reaction is explicitly simulated)

In summary, in chemical systems formed by living cells, the small numbers of molecules of a few reactant species can results in dynamical behavior that is discrete and stochastic, rather than continuous and deterministic. By discrete, we mean the integer-valued nature of small molecular populations, which makes their representation by real-valued (continuous) variables inappropriate. By stochastic, we mean the random behavior that arises from the lack of total predictability in molecular dynamics.

1.4 Notions of Combinatorics

As we have seen in the previous Section, the probability that a reaction R_μ will occur in a solution is necessary to know the number of distinct R_μ molecular reactant combinations and then we have to deal with combinatorics.

To count the number of occurrences of the set of reactants $\ell_1 R_1 + \ldots + \ell_k R_k$ in a solution in which each reactants R_i is present in number $p_i \geq \ell_i$, it is necessary to calculate the number of *simple combinations*, computing $\prod_{i=1}^{k} \binom{p_i}{\ell_i}$ where $\binom{n}{k}$ is the *binomial coefficient*. This coefficient can be computed by the Newton's generalized binomial theorem as

$$\left(\begin{array}{c} n \\ k \end{array} \right) = \frac{n!}{k!\,(n-k)!} = \frac{1}{k!} \prod_{r=0}^{k-1} (n-r) = \frac{n(n-1)(n-2)\ldots(n-(k-1))}{k!}$$

Example 1.4. Having a solution like (with + we indicate the presence of elements in a uniform distributed solution)

$$100A + 2B + 60C + 40D + E$$

and a reaction that requires in its left hand side the presence of the following set of reactants

$$50A + B + C + D + E$$

the number of occurrences of the reactants in the solutions is computed as follows

$$\binom{100}{50} * \binom{2}{1} * \binom{60}{1} * \binom{40}{30} * \binom{1}{1} = 9.381946092840906 * 10^{29}$$

that is: the reaction can happens in $\backsim 9 * 10^{29}$ different ways.

Note that this combinatorics calculations require to handle very big numbers (for example $30! = 265252859812191058636308480000000$) and the calculation of factorials are very expensive.

1.5 Rule Based Systems and Term Rewriting Systems

Rule Based Systems

Rule based systems (also called *production systems*) are often used as a natural way to express models with a certain behavior. In fact, they are often used like notation to formally specify as productions (= rules) the allowable behaviors of a system, just as a grammar is intended to define the legal sentences in a language. In fact, in contrast to a procedural computation, in which knowledge about the problem domain is mixed in with instructions about the flow of control, a rule based engine model allows a more complete separation of the knowledge (in the rules) from the control (the inference engine).

The basic structure of a rule based systems (and their associated interpreter) is quite simple. In its most essential form a production system consists of two interacting data structures, connected through a simple processing cycle :

- A *working memory* consisting of a collection of symbolic data items called working memory elements.
- A *production memory* consisting of condition-action rules called *productions*, whose conditions describe configurations of element that might appear in working memory and whose actions specify modifications to the contents of working memory.

Production memory and working memory are related through the *recognize-act* cycle. This consists of three distinct stages:

1. The *match* process, which finds productions whose conditions match against the current state of working memory; the same rule may match against memory in different ways, each such mapping is called *instantiation*.
2. The *conflict resolution* process, which selects one or more of the instantiated productions for applications.
3. The *act* process which applies the instantiated actions of the selected rules, thus modifying the contents of working memory.

The basic recognize-act process operates in cycles, with one or more rules being selected and applied, the new constants of memory leading another set of rules to be applied, and so on. This cycling continues until no rules are matched or until an explicit halt command is encountered. It starts with a rule-base, which contains all of the appropriate knowledge encoded into *If–Then* rules, and a working memory, which may or may not initially contain any data, assertions or initially known information. The system examines all the rule conditions (*if*) and determines a subset, the *conflict set*[1], of the rules whose conditions are satisfied based on the working memory. Of this conflict set, one of those rules is triggered (fired). Which one is chosen is based on a conflict resolution strategy. When the rule is fired, any actions specified in

[1]Conflict set is also called *agenda* in the within of production systems

its *then* clause are carried out. These actions can modify the working memory, the rule-base itself, or do just about anything else the system programmer decides to include. This loop of firing rules and performing actions continues until one of two conditions are met: there are no more rules whose conditions are satisfied or a rule is fired whose action specifies the program should terminate.

Term Rewriting Systems

Term rewriting systems are a subclass of rule based systems where the state and the rule are made of trees. Tree terms are states of an abstract machine, while rewriting rules are state transforming functions. Like in productions systems, in this framework a computations simply are sequence of rewrites. Term rewriting systems have numerous practical applications like formal reasoning tool (as for example cryptographic protocol analysis, hardware design verification), theorem proving, computer algebra, language optimization etc... See [51, 34, 14, 60] for a widen look on this field.

Part II

METHODS

Chapter 2

Stochastic Calculus of Looping Sequences

In this chapter we introduce the *Calculus of Looping Sequences* (CLS), a formalism based on term rewriting suitable for describing qualitative aspects of biological and biochemical systems.

Being a formalism based on the rewriting of terms, we start introducing the syntax of its valid terms (2.1.1) and the semantics on which the calculation is forwarded (2.1.2).

We then examine the stochastic extension of CLS, namely the *Stochastic Calculus of Looping Sequences* (2.2), that is useful in order to quantitatively simulate these systems. We present how the CLS semantics is extended (2.2.1) and how the stochastic semantics can be simulated (2.2.2).

Finally in (2.2.2) we give a brief discussion about the applications of CLS in biological modeling.

2.1 Calculus of Looping Sequences

In this chapter we recall the formalism for the description of biological systems based on term rewriting that allows describing (at least) proteins, DNA fragments, membranes and macromolecules in general. We want not ignoring the physical structure of these elements keeping the syntax and the semantics of the formalism as simple as possible.

CLS is based on term rewriting, and hence a CLS model consists of a term and a set of rewrite rules. The term represents the structure of the modeled system, and the rewrite rules represent the ways in which it can evolves.

The kind of structures that most frequently appear in cellular components is probably the sequence. A DNA fragment, for instance, is a sequence of nucleic acids, and it can be seen, at a higher level of abstraction, also as a sequence of genes. Proteins are sequences of amino acids, and they can be seen also as sequences of interaction sites. Membrane, instead, are essentially closed surfaces interspersed with proteins and molecules of various kinds, hence we can see them abstractly as closed circular sequences whose elements or subsequences describe the entities that

are placed in the membrane surface. Finally, there are usually many components in a biological system, some of which may be contained in some membranes, and membranes may be nested in various ways, thus forming a hierarchical structure that may change over time.

In the rest of the chapter we examine in details the CLS formalism that is based on term rewriting and which tries to fulfill these requirements.

2.1.1 Syntax

We start with defining the syntax of valid terms.

As already said before, we have to define terms able to describe (i) sequences, (ii) which may be closed and may contain something, and (iii) which may be juxtaposed to other sequences in the system. For each of these three points we define an operator in the grammar of terms. Moreover, we assume a possibly infinite alphabet of elements $\mathcal{E}$ ranged over by $a, b, c, \ldots$ to be used as the building blocks of terms, and a neutral element ε representing the empty term. Terms of the calculus are defined as follows.

Definition 2.1 (***Terms***)**.** *Terms* T and *Sequences* S of CLS are given by the following grammar:

$$\begin{array}{rcl} T & ::= & S \;\mid\; \left(S\right)^{L}\rfloor T \;\mid\; T \,|\, T \\ S & ::= & \varepsilon \;\mid\; a \;\mid\; S \cdot S \end{array}$$

where a is a generic element of $\mathcal{E}$. We denote with $\mathcal{T}$ the infinite set of terms, and with $\mathcal{S}$ the infinite set of sequences.

Terms include the elements in the alphabet $\mathcal{E}$ and the empty term ε. Moreover, the following operators can be used to build more complex terms:

- Sequencing (or concatenation) $\_ \cdot \_$: creates a ordinated sequence whose elements are elements of the alphabet $\mathcal{E}$ or in its turn sequences.
- Looping and Containment $\left(\_\right)^{L}\rfloor \_$: creates a closed circular sequence of the sequence to which it is applied. The operator is called looping because, as we shall see, it is always possible to rotate the representation of the circular sequence. Also represents the containment of the second term to which it is applied into the membrane made by the looping sequence. Note that either the membrane or its content con be ε.
- Parallel composition $\_ | \_$: represents the juxtaposition of the two terms to which it is applied.

A term can be a sequence, or a looping sequence containing a term, or the parallel composition of two terms.

Brackets can be used to indicate the order of application of the operators, and we assume $\left(\_\right)^{L}\rfloor \_$ to have the precedence over $\_ | \_$. In Figure 2.1 we show some examples of CLS terms and their visual representation.

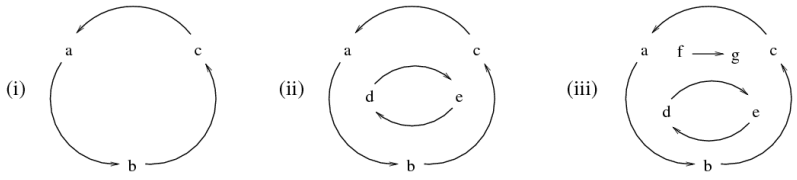


Figure 2.1: (i) represents $\big(a \cdot b \cdot c\big)^L$;
(ii) represents $\big(a \cdot b \cdot c\big)^L \rfloor \big(d \cdot e\big)^L$;
(iii) represents $\big(a \cdot b \cdot c\big)^L \rfloor (\big(d \cdot e\big)^L \mid f \cdot g)$.

On terms and sequences is defined the notion of *structural congruence*: it is a relation used to consider as equal terms that are syntactically different but representing the same process, usually states associativity and commutativity of operators.

Definition 2.2 (***Structural Congruence***)**.** The structural congruence relations $\equiv_S$ and $\equiv_T$ are the least congruence relations on sequences and on terms, respectively, satisfying the following rules:

$$S_1 \cdot (S_2 \cdot S_3) \equiv_S (S_1 \cdot S_2) \cdot S_3 \qquad S \cdot \varepsilon \equiv_S \varepsilon \cdot S \equiv_S S$$

$$S_1 \equiv_S S_2 \text{ implies } S_1 \equiv_T S_2 \text{ and } \big(S_1\big)^L \rfloor T \equiv_T \big(S_2\big)^L \rfloor T$$
$$T_1 \mid T_2 \equiv_T T_2 \mid T_1 \qquad T_1 \mid (T_2 \mid T_3) \equiv_T (T_1 \mid T_2) \mid T_3 \qquad T \mid \varepsilon \equiv_T T$$
$$\big(\epsilon\big)^L \rfloor \varepsilon \equiv \varepsilon \qquad \big(S_1 \cdot S_2\big)^L \rfloor T \equiv_T \big(S_2 \cdot S_1\big)^L \rfloor T$$

Rules of the structural congruence state the associativity of $\cdot$ and $\mid$, the commutativity of the latter and the neutral role of ε. Moreover, axiom $\big(S_1 \cdot S_2\big)^L \rfloor T \equiv_T \big(S_2 \cdot S_1\big)^L \rfloor T$ says that elementary sequences in a looping can rotate. We remark that we have $\big(\varepsilon\big)^L \rfloor T \not\equiv T$ if $T \not\equiv \varepsilon$, hence $\big(\varepsilon\big)^L$ does not play a neutral role if it is not empty. In the following, for simplicity, we will use $\equiv$ in place of $\equiv_T$ or $\equiv_S$.

Now we define rewrite rules, which can be used to describe the evolution of terms. Roughly, a rewrite rule is a triple consisting of two terms and one condition to be satisfied. The two terms describe what term the rule can be applied to and the term obtained after the application of the rule, respectively, and the condition must be satisfied before applying the rule.

In order to allow a rule to be applied to a wider range of terms, we introduce typed variables in the terms of a rule. This allows to express rule schemata, instead of single rules.

Patterns in CLS include three different types of variables: two are associated with the two different syntactic categories of terms and sequences, and one is associated with single alphabet elements. We assume a set of term variables TV ranged over by $X, Y, Z, \ldots$, a set of sequence variables SV ranged over by $\widetilde{x}, \widetilde{y}, \widetilde{z}, \ldots$, and a set of element variables $\mathcal{X}$ ranged over by $x, y, z, \ldots$. All these sets are possibly infinite and pairwise disjoint. We denote by $\mathcal{V}$ the set of all variables, $\mathcal{V} = TV \cup SV \cup \mathcal{X}$.

Definition 2.3 (***Patterns***)**.** *Patterns* P and *sequence patterns* SP of CLS are given by the following grammar:

$$\begin{array}{rcl} P & ::= & SP \;\mid\; \left(SP\right)^{L} \rfloor T \;\mid\; T \,|\, T \;\mid\; X \\ SP & ::= & \varepsilon \;\mid\; a \;\mid\; SP \cdot SP \;\mid\; \widetilde{x} \;\mid\; x \end{array}$$

where a is a generic element of $\mathcal{E}$, and $X, \widetilde{x}$ and x are generic elements of TV, SV and $\mathcal{X}$, respectively. We denote with $\mathcal{P}$ the infinite set of patterns.

We assume the structural congruence relation to be trivially extended to patterns.

To obtain a ground term from a pattern we introduce the concept of *instantiation*. An instantiation is a partial function $\sigma : \mathcal{V} \to \mathcal{T}$. An instantiation must preserve the type of variables, thus for $X \in TV$, $\widetilde{x} \in SV$ and $x \in \mathcal{X}$ we have $\sigma(X) \in \mathcal{T}, \sigma(\widetilde{x}) \in \mathcal{S}$ and $\sigma(x) \in \mathcal{E}$, respectively. Given $P \in \mathcal{P}$, with $P\sigma$ we denote the term obtained by replacing each occurrence of each variable $X \in \mathcal{V}$ appearing in P with the corresponding term $\sigma(X)$. With Σ we denote the set of all the possible instantiations and, given $P \in \mathcal{P}$, with $Var(P)$ we denote the set of variables appearing in P. Finally, we define a function $occ : \mathcal{E} \times \mathcal{T} \to \mathbb{N}$ such that $occ(a, T)$ returns the number of the elements a syntactically occurring in the term T.

Now we define rewrite rules; that are defined essentially as pairs of terms, in which the first term describes the portion of the system in which the event modeled by the rule may occur, and the second term describes how that portion of the system changes when the event occurs. In the terms of a rewrite rule we allow the use of variables. As a consequence, a rule will be applicable to all terms which can be obtained by properly instantiating its variables.

Definition 2.4 (***Rewrite Rules***)**.** A rewrite rule is a pair of patterns (P_1, P_2), denoted with $P_1 \mapsto P_2$, where $P_1, P_2 \in PP$, $P_1 \not\equiv \varepsilon$ and such that $Var(P_2) \subseteq Var(P_1)$. We denote with $\Re$ the infinite set of all the possible rewrite rules. We say that a rewrite rule is *ground* if $Var(P_1) = Var(P_2) = \varnothing$, and a set of rewrite rules $\mathcal{R} \in Re$ is *ground* if all the rewrite rules it contains are ground.

A rewrite rule (P_1, P_2) states that a term $P_1\sigma$, obtained by instantiating variables in P_1 by some instantiation function σ, can be transformed into the ground term $P_2\sigma$. Rule application is the mechanism of evolution of CLS terms.

2.1.2 Semantics

We define the semantics of CLS as a transition system, in which states corresponds to terms, and transitions corresponds to rule applications.

The semantics of CLS is defined by resorting to the notion of *contexts*.

Definition 2.5 (***Contexts***)**.** *Contexts* $\mathcal{C}$ are given by the following grammar:

$$\mathcal{C} ::= \Box \;\mid\; \mathcal{C} \,|\, T \;\mid\; T \,|\, \mathcal{C} \;\mid\; \left(S\right)^{L} \rfloor \mathcal{C}$$

where $T \in \mathcal{T}$ and $S \in \mathcal{S}$. Context $\Box$ is called the *empty context*.

By definition, every context contains a single $\Box$. Let us assume $C, C' \in \mathcal{C}$. With $C[T]$ we denote the term obtained by replacing $\Box$ with T in C; with $C[C']$ we denote the context composition, whose result is the context obtained by replacing $\Box$ with C' in C. The structural congruence relation can be easily extended to contexts, namely $C \equiv C'$ if and only if $C[\varepsilon] \equiv C'[\varepsilon]$.

A rewrite rule $T \mapsto T'$ states that a ground term $T\sigma$, obtained by instantiating variables in T by some instantiation function σ, can be transformed into the ground term $T'\sigma$. The rewrite rules can be applied to terms only if they occur in a legal context.

Note that the general form of rewrite rules does not allows to have sequences as contexts. A rewrite rule introducing a parallel composition on the right hand side (as in $a \mapsto b \,|\, c$) applied to an element of a sequence (for example $m \cdot a \cdot m$) would result in a syntactically incorrect term (in this case $m \cdot (b \,|\, c) \cdot m$).

To modify a sequence, the whole sequence must appears in the left hand side of the rule. For example, rule $a \cdot b \cdot c \mapsto a \,|\, b \cdot c$ represents the separation of a from the sequence $a \cdot b \cdot c$. Such a rule, however, can be applied only on the exact sequence $a \cdot b \cdot c$. Variables in the left hand side of the rule allow the application of the rule to a class of sequences. For example, rule $a \cdot \widetilde{x} \mapsto a \,|\, \widetilde{x}$ can be applied to any sequence starting with element a, and hence, the term $a \cdot b$ can be rewritten as $a \,|\, b$, and the term $a \cdot b \cdot c$ can be rewritten as $a \,|\, b \cdot c$.

The reduction semantics of CLS is defined as a transition system as follows.

Definition 2.6 (***Reduction Semantics***)**.** Given a finite set of rewrite rules $\mathcal{R}$, the *reduction semantics* of CLS is the least relation closed with respect to $\equiv$ and satisfying the following inference rule:

$$\frac{T \mapsto T' \in \mathcal{R} \qquad T\sigma \not\equiv \varepsilon \qquad \sigma \in \Sigma \qquad C \in \mathcal{C}}{C[T\sigma] \to C[T'\sigma]}$$

A *model* in CLS is given by a term describing the initial state of the modeled system and by a set of rewrite rules describing all the possible events that may occur in the system. That is :

Definition 2.7 (***CLS Model***)**.** A SCLS model is a pair $(T_0, \mathcal{R})$ where T_0 it is the starting state of the systems, and $\mathcal{R}$ is a finite set of rule schema.

2.2 Stochastic Calculus of Looping Sequences

To focus on quantitative aspects of our formalism, in particular, to model speed of activities, it has been developed a stochastic extension of CLS, called *Stochastic Calculus of Looping Sequence* (SCLS for short), that incorporate the stochastic framework developed by Gillespie in [44]. Rates are associated with rewrite rules in order to model the speed of the described activities. Therefore, transitions derived in SCLS are driven by a rate that models the parameter of an exponential distribution and characterizes the stochastic behavior of the transition. The choice of the next rule to be applied and of the time of its application is based on the classical Gillespies algorithm [44].

We now introduce the concept of an enriched form of rewrite rule: the *rewrite rule schema*. Differently to a rewrite rule, a rule schema has a *rate function* instead of a rate constant. This corresponds to express an infinite set of rule, each of which has a possible different rate constant obtained applying the rate function to the instantiation of the variables in the rule that brings from the rule with variable to correspondent ground rule.

Definition 2.8 (***Stochastic Rewrite Rule Schema***)**.** A *stochastic rewrite rule schema* is a triple (P, P', f), denoted with $P \overset{f}{\mapsto} P'$, where $P, P' \in \mathcal{P}$, $P \not\equiv \varepsilon$ and such that $Var(P') \subseteq Var(P)$, and $f : \Sigma \to \mathbb{R}^{\geq 0}$ is the *rewriting rate function*.

Patterns of a stochastic rewrite rule schema may contain variables. The rewriting rate function may depend on the instantiations of the variables appearing in the left hand side term of the schema. By instantiating the variables, a *stochastic ground rewrite rule* with a *rewriting rate constant* is obtained.

Definition 2.9 (***Stochastic Ground Rewrite Rule***)**.** Given a stochastic rewrite rule schema $R = (T, T', f)$ and an instantiation $\sigma \in \Sigma$, the *ground rewrite rule* derived from R and σ is a triple (Tg, Tg', p), denoted $Tg \overset{p}{\mapsto} Tg'$, where $Tg = T\sigma$, $Tg' = T'\sigma$, and $p = f(\sigma)$ is the *rewriting rate constant*.

Example 2.10. Let us assume a function $occ : \mathcal{E} \times \mathcal{T} \to \mathbb{N}$ such that $occ(a, T)$ returns the number of elements a syntactically occurring in the term T. Consider the rewrite rule schema $R = (a \,|\, \big(c \cdot \widetilde{x}\big)^L, b \,|\, \big(\widetilde{x}\big)^L, f(\sigma) = occ(c, \sigma(\widetilde{x})) + 1)$ and the instantiation $\sigma(\widetilde{x}) = b \cdot c$. We obtain a stochastic ground rewrite rule $(a \,|\, \big(c \cdot b \cdot c\big)^L, b \,|\, \big(b \cdot c\big)^L, p)$, where $p = f(\sigma) = occ(c, b \cdot c) + 1 = 2$.

Definition 2.11 (***Applicable Ground Rewrite Rules***)**.** Given a rewrite rule schema $R = T \overset{f}{\mapsto} T'$ and a term $T_0 \in \mathcal{T}$, the set of *ground rewrite rules* derived from R and applicable to T_0 is defined as

$$AR(R, T_0) = \{Tg \overset{p}{\mapsto} Tg' \,|\, \exists \sigma \in \Sigma, C \in \mathcal{C}.\ Tg = T\sigma, Tg' = T'\sigma, T_0 \equiv C[Tg], p = f(\sigma)\}.$$

Given a finite set of rewrite rule schemata $\mathcal{R}$ and a term $T_0 \in \mathcal{T}$, the set of *ground rewrite rules* derived from $\mathcal{R}$ and applicable to T_0 is the set:

$$AR(\mathcal{R}, T_0) = \bigcup_{R \in \mathcal{R}} AR(R, T_0).$$

In the following we will need also to know from which context each reactant has been extracted, hence we define the *multiset of extracted reactants* of T as the multiset of all the pairs (T', C) where $T' \not\equiv \varepsilon$ is a reactant in T and C is the context such that $C[T'] \equiv T$. In the definition we use the auxiliary function $\circ : \mathcal{C} \times (\mathbb{N} \times \mathcal{T} \times \mathcal{C}) \mapsto (\mathbb{N} \times \mathcal{T} \times \mathcal{C})$ defined as $C \circ (i, T, C') = (i, T, C[C'])$ extended to multisets of triples over $\mathbb{N} \times \mathcal{T} \times \mathcal{C}$ in the obvious way.

Definition 2.12 (***Multiset of Extracted Reactants***)**.** Given a term $T \in \mathcal{T}$, the multiset of reactants extracted from T is defined as

$$ext(T) = \left\{(T', C) | (n, T', C) \in ext_\ell(0, T)\right\}$$

where ext_ℓ is given by the following recursive definition:

$$\begin{aligned}
ext_\ell(i, S) &= \{(i, S, \Box)\} \\
ext_\ell(i, \left(S\right)^L) &= \{(i, \left(S\right)^L, \Box)\} \\
ext_\ell(i, \left(S\right)^L \rfloor T') &= \{(i, \left(S\right)^L \rfloor T', \Box)\} \ \cup \ \left(S\right)^L \rfloor \Box \circ ext_\ell(i+1, T') \\
ext_\ell(i, T_1 \,|\, T_2) &= T_2 \,|\, \Box \circ ext_\ell(i, T_1) \ \cup \ T_1 \,|\, \Box \circ ext_\ell(i, T_2) \\
&\quad \cup \ \{(i, T_1^e \,|\, T_2^e, C_1^e[C_2^e]) \,|\, (i, T_j^e, C_j^e) \in ext_\ell(i, T_j), j \in \{1, 2\}\}
\end{aligned}$$

Given a term $T \in \mathcal{T}$, $ext(T)$ extracts from T the multiset of reactants to which a rewrite rule could be applied. Each element of the multiset contains also the context from which each reactant is extracted. Because a reactant is an occurrence of a sub term in T, we have, for example, $ext(a \,|\, a) = \{(a, a \,|\, \Box), (a, a \,|\, \Box), (a \,|\, a, \Box)\}$, where the two $(a, a \,|\, \Box)$-elements correspond to the two reactants a in $a \,|\, a$. The function ext makes use of the function ext_ℓ in order to separate reactants obtained at different levels of containment (which cannot be mixed).

Example 2.13. Let T be $a \,|\, \left(b\right)^L \rfloor c$, then
$ext_\ell(0, T) = \{(0, a, \Box \,|\, \left(b\right)^L \rfloor c), (0, \left(b\right)^L \rfloor c, a \,|\, \Box), (1, c, a \,|\, \left(b\right)^L \rfloor \Box), \ (0, T, \Box)\}$,
and
$ext(T) = \{(a, \Box \,|\, \left(b\right)^L \rfloor c), (\left(b\right)^L \rfloor c, a \,|\, \Box), (c, a \,|\, \left(b\right)^L \rfloor \Box), (T, \Box)\}$. Note that ext_ℓ avoids $(a \,|\, c, C)$ to be extracted from T, for any context C.

2.2.1 Semantics of Stochastic CLS

The ext function computes the multiset of reactants of a term. We use such a function to compute the application rate of a ground rewrite rule. In particular, we compute the *application cardinality* of the rule, that is the number of reactants in the term in which the rule is applied that are equivalent to the left–hand side of the rule. This value will be multiplied by the kinetic constant of the reaction to obtain the application rate.

Definition 2.14 (***Application Cardinality***)**.** Given a ground rewrite rule $R = T_2 \overset{c}{\mapsto} T_2$ and two terms $T, T_r \in \mathcal{T}$, the *application cardinality* of rule R leading from T to T_r, $AC(R, T, T_r)$, is defined as follows:

$$AC(R, T, T_r) = |\left\{(T', C) \in ext(T) \text{ such that } T' \equiv T_1 \wedge C[T_2] \equiv T_r\right\}|.$$

As already mentioned, given a term T, a ground rewrite rule can be applied to different reactants of T. Hence, according to the reactants to which the rule is applied, the rewrite of T may result in different terms. For any reachable term, the application cardinality counts the number of reactants leading to it.

Example 2.15. Consider the ground rewrite rule $R = a \overset{c}{\mapsto} b$ and term $T = a \,|\, a \,|\, \left(m\right)^L \rfloor\, a$. The left part of the rule, consisting of the single element a, is contained three times in the set $ext(T)$, however the application of rule R in those three points gives rise to different terms. In particular, for the two elements (a, C) where $C = a \,|\, \left(m\right)^L \rfloor\, a \,|\, \Box$ we have $T_r = C[T_2] = a \,|\, \left(m\right)^L \rfloor\, a \,|\, b$, and hence $AC(R, T, T_r) = 2$. On the other hand, for the element (a, C'), with $C' = a \,|\, a \,|\, \left(m\right)^L \rfloor\, \Box$, we have $T'_r = C'[T_2] = a \,|\, a \,|\, \left(m\right)^L \rfloor\, b$, and hence $AC(R, T, T'_r) = 1$.

We now give the semantics of Stochastic CLS.

Definition 2.16 (***Semantics***)**.** Given a finite set of rewrite rule schemata $\mathcal{R}$, the semantics of Stochastic CLS is the least labeled transition relation satisfying the following inference rule:

$$\frac{R = T_1 \overset{c}{\mapsto} T_2 \in AR(\mathcal{R}, T) \qquad T \equiv C[T_1]}{T \xrightarrow{R, c \cdot AC(R,T,C[T_2])} C[T_2]}$$

The stochastic reduction semantics associates with each transition a rate which is the parameter of an exponential distribution that characterizes the stochastic behavior of the activity corresponding to the rewrite rule applied. The rate is obtained as the product of the rewriting rate constant and the application cardinality of the rule. The rewriting rate constant, obtained by instantiating the rewriting rate function of the schema from which the ground rewrite rule derives, expresses the contribution of the chosen instantiation, and the application cardinality expresses the number of reactants to which the rule can be applied and which give the same result. The higher is this value, the higher is the rate of the transition.

The stochastic reduction semantics is essentially a *stochastic automaton*[1], that is an automaton of which transitions are labeled with rates.

2.2.2 Stochastic Simulation

Given an SCLS model to simulate, an automaton (that is a transition system) is given by the stochastic reduction semantics; thus we can follow a standard simulation procedure that corresponds to Gillespie's simulation algorithm for chemical reactions [44]. Roughly speaking, the algorithm starts from the initial state of the automaton and performs a sequence of steps by moving from state to state. At each step a global clock variable (initially set to t_0) is incremented by a random quantity which is exponentially distributed with the exit rate of the current state s as parameter, and the next state s' is randomly chosen with a probability proportional to the rate of the transitions multiplied by the relative application cardinality.

However, the whole automaton describing the systems has often a huge number of states, hence its construction for simulation is practically unfeasible. Simulations of single evolutions of the systems, instead, can be performed without building the whole transition system. In fact, in order to perform simulations we can follow a procedure that corresponds to Gillespie's simulation algorithm for chemical reactions

[1] More precisely is a *Continuous Time Markov Chain* as discussed in [77].

[44]. A state of the simulation is a pair (T, t) where T is the current term and $t \in \mathbb{R}^{\geq 0}$ is the global clock. Assuming a finite set of rewrite rule schemata $\mathcal{R}$ and an initial term T_0, the initial state of the simulation is the pair $(T_0, 0)$.

Given a simulation state (T, t), from the stochastic reduction semantics, we have a finite set of transitions starting from T, namely the set of transitions $\{T \xrightarrow{Rg_i, r_i} T_i\}$, with $i \in [1, n]$, where r_i gives the rate of the i-th transition, and n is the number of transitions starting from T. Note that different transitions can be labeled by the same rewrite rule. Now, a simulation step transforms the state (T, t) into $(T_i, t + \tau)$ where τ is exponentially distributed with parameter $E = \sum_{i=1}^{n} r_i$ and i is chosen with probability $\frac{r_i}{E}$.

CLS as an Abstraction for Biomolecular Systems

An abstraction is a mapping from a real–world domain to a mathematical domain, that may allow to highlight some essential properties of a system while ignoring other, complicating, ones. In [88], Regev and Shapiro show how to abstract bimolecular systems as concurrent computation by identifying the bimolecular entities and events of interest and by associating them with concepts of concurrent computation such as concurrent processes and communications.

The use of rewrite systems, such as CLS, to describe biological systems is founded on a different abstraction. Usually, entities (and their structures) are abstracted by terms of the rewrite system, and events by rewriting rules.

The guidelines for the abstraction of bimolecular entities and events into CLS are given in Table 2.1 and Table 2.2. More details about how to use CLS as an abstraction for bimolecular systems can be found in the Chapter 4 of Ph.D. thesis of Milazzo [77].

Biomolecular Entity	**CLS Term**
Elementary object (genes, domains, other molecules, etc...)	Alphabet symbol
DNA strand	Sequence of elements representing genes
RNA strand	Sequence of elements representing transcribed genes
Protein	Sequence of elements representing domains or single alphabet symbol
Molecular population	Parallel composition of molecules
Membrane	Looping sequence

Table 2.1: Guidelines for the abstraction of biomolecular entities into CLS.

Biomolecular Event	**Examples of CLS Rewrite Rule**
State change	$a \mapsto b$ $\widetilde{x} \cdot a \cdot \widetilde{y} \mapsto \widetilde{x} \cdot b \cdot \widetilde{y}$
Complexation	$a \mid b \mapsto c$ $\widetilde{x} \cdot a \cdot \widetilde{y} \mid b \mapsto \widetilde{x} \cdot c \cdot \widetilde{y}$
Decomplexation	$c \mapsto a \mid b$ $\widetilde{x} \cdot c \cdot \widetilde{y} \mapsto \widetilde{x} \cdot a \cdot \widetilde{y} \mid b$
Catalysis	$c \mid P_1 \mapsto c \mid P_2$ where $P_1 \mapsto P_2$ is the catalyzed event
State change on membrane	$\left(a \cdot \widetilde{x}\right)^L \rfloor X \mapsto \left(b \cdot \widetilde{x}\right)^L \rfloor X$
Complexation on membrane	$\left(a \cdot \widetilde{x} \cdot b \cdot \widetilde{y}\right)^L \rfloor X \mapsto \left(c \cdot \widetilde{x} \cdot \widetilde{y}\right)^L \rfloor X$ $a \mid \left(b \cdot \widetilde{x}\right)^L \rfloor X \mapsto \left(c \cdot \widetilde{x}\right)^L \rfloor X$ $\left(b \cdot \widetilde{x}\right)^L \rfloor (a \mid X) \mapsto \left(c \cdot \widetilde{x}\right)^L \rfloor X$
Decomplexation on membrane	$\left(c \cdot \widetilde{x}\right)^L \rfloor X \mapsto \left(a \cdot b \cdot \widetilde{x}\right)^L \rfloor X$ $\left(c \cdot \widetilde{x}\right)^L \rfloor X \mapsto a \mid \left(b \cdot \widetilde{x}\right)^L \rfloor X$ $\left(c \cdot \widetilde{x}\right)^L \rfloor X \mapsto \left(b \cdot \widetilde{x}\right)^L \rfloor (a \mid X)$
Catalysis on membrane	$\left(c \cdot \widetilde{x} \cdot SP_1 \cdot \widetilde{y}\right)^L \mapsto \left(c \cdot \widetilde{x} \cdot SP_2 \cdot \widetilde{y}\right)^L$ where $SP_1 \mapsto SP_2$ is the catalyzed event
Membrane crossing	$a \mid \left(\widetilde{x}\right)^L \rfloor X \mapsto \left(\widetilde{x}\right)^L \rfloor (a \mid X)$ $\left(\widetilde{x}\right)^L \rfloor (a \mid X) \mapsto a \mid \left(\widetilde{x}\right)^L \rfloor X$ $\widetilde{x} \cdot a \cdot \widetilde{y} \mid \left(\widetilde{z}\right)^L \rfloor X \mapsto \left(\widetilde{z}\right)^L \rfloor (\widetilde{x} \cdot a \cdot \widetilde{y} \mid X)$ $\left(\widetilde{z}\right)^L \rfloor (\widetilde{x} \cdot a \cdot \widetilde{y} \mid X) \mapsto \widetilde{x} \cdot a \cdot \widetilde{y} \mid \left(\widetilde{z}\right)^L \rfloor X$
Catalyzed membrane crossing	$a \mid \left(b \cdot \widetilde{x}\right)^L \rfloor X \mapsto \left(b \cdot \widetilde{x}\right)^L \rfloor (a \mid X)$ $\left(b \cdot \widetilde{x}\right)^L \rfloor (a \mid X) \mapsto a \mid \left(b \cdot \widetilde{x}\right)^L \rfloor X$ $\widetilde{x} \cdot a \cdot \widetilde{y} \mid \left(b \cdot \widetilde{z}\right)^L \rfloor X \mapsto \left(b \cdot \widetilde{z}\right)^L \rfloor (\widetilde{x} \cdot a \cdot \widetilde{y} \mid X)$ $\left(b \cdot \widetilde{z}\right)^L \rfloor (\widetilde{x} \cdot a \cdot \widetilde{y} \mid X) \mapsto \widetilde{x} \cdot a \cdot \widetilde{y} \mid \left(b \cdot \widetilde{z}\right)^L \rfloor X$
Membrane joining	$\left(\widetilde{x}\right)^L \rfloor (a \mid X) \mapsto \left(a \cdot \widetilde{x}\right)^L \rfloor X$ $\left(\widetilde{x}\right)^L \rfloor (\widetilde{y} \cdot a \cdot \widetilde{z} \mid X) \mapsto \left(\widetilde{y} \cdot a \cdot \widetilde{z} \cdot \widetilde{x}\right)^L \rfloor X$
Catalyzed membrane joining	$\left(b \cdot \widetilde{x}\right)^L \rfloor (a \mid X) \mapsto \left(a \cdot b \cdot \widetilde{x}\right)^L \rfloor X$ $\left(\widetilde{x}\right)^L \rfloor (a \mid b \mid X) \mapsto \left(a \cdot \widetilde{x}\right)^L \rfloor (b \mid X)$ $\left(b \cdot \widetilde{x}\right)^L \rfloor (\widetilde{y} \cdot a \cdot \widetilde{z} \mid X) \mapsto \left(\widetilde{y} \cdot a \cdot \widetilde{z} \cdot \widetilde{x}\right)^L \rfloor X$ $\left(\widetilde{x}\right)^L \rfloor (\widetilde{y} \cdot a \cdot \widetilde{z} \mid b \mid X) \mapsto \left(\widetilde{y} \cdot a \cdot \widetilde{z} \cdot \widetilde{x}\right)^L \rfloor (b \mid X$
Membrane fusion	$\left(\widetilde{x}\right)^L \rfloor (X) \mid \left(\widetilde{y}\right)^L \rfloor (Y) \mapsto \left(\widetilde{x} \cdot \widetilde{y}\right)^L \rfloor (X \mid Y)$
Catalyzed membrane fusion	$\left(a \cdot \widetilde{x}\right)^L \rfloor (X) \mid \left(b \cdot \widetilde{y}\right)^L \rfloor (Y) \mapsto \left(a \cdot \widetilde{x} \cdot b \cdot \widetilde{y}\right)^L \rfloor (X \mid Y)$
Membrane division	$\left(\widetilde{x} \cdot \widetilde{y}\right)^L \rfloor (X \mid Y) \mapsto \left(\widetilde{x}\right)^L \rfloor (X) \mid \left(\widetilde{y}\right)^L \rfloor (Y)$
Catalyzed membrane division	$\left(a \cdot \widetilde{x} \cdot b \cdot \widetilde{y}\right)^L \rfloor (X \mid Y) \mapsto \left(a \cdot \widetilde{x}\right)^L \rfloor (X) \mid \left(b \cdot \widetilde{y}\right)^L \rfloor (Y)$

Table 2.2: Guidelines for the abstraction of biomolecular events into CLS.

Chapter 3

Development of a Stochastic Simulator for CLS

"The software is hard" Donald Knuth

In this chapter we present all the issues we have found in the development and implementation of a stochastic simulator for CLS. Firstly we introduce problems that we have faced (3.1.1); we present our choices (3.2.1) and the architecture that we have designed for this kind of simulator (3.2.2). Then we look more in details at the data structures we used (3.3.1), at the developed algorithm for CLS pattern matching (3.3.2) and finally at the extension of the Gillespie's algorithm that we have made in order to take account of rule schemata (3.3.3).

3.1 Problems

3.1.1 Goals

The main goal of the work of this thesis is the development of a stochastic simulator for SCLS, allowing to express rule schemata through the use of typed variables and rate functions. The simulator have therefore the goal of make efficient real–time simulations, involving million of iterations, allowing to plot the amounts of each element in the population together with the amounts of some user defined pattern and to save the simulation evolution to file system, as text, html or spreadsheet.

With user defined pattern concentrations we mean that the user wants to monitor the concentration of some elements in a certain position in the hierarchy of the state term. This information it is not visible by the plot of the concentration of single elements and thus it is necessary to express what the user wants to monitor through SCLS patterns. Although this problem is not closely related to the simulation problem we have to take care of it, delegating the counting of such patterns to the pattern matching algorithm.

Example 3.1. The rewrite rule schema

$$a \mid \left(b\right)^{L} \rfloor (X) \overset{k}{\mapsto} \left(b\right)^{L} \rfloor (a \mid X)$$

asserts that a membrane made of a b element can take inside some a elements eventually present in the same solution (expressed through the parallel composition operator). The user could wants to simulate this behavior starting in a state with a certain number n of a, observing the speed in which the a go inside the membrane. This in not observable if we plot simply the amount of a in the solution: it will remains constant at its initial quantity n.

To observe the amount of a inside the membrane we can observe the value given by the multiplication of the number of occurrences of the CLS pattern $\left(b\right)^{L} \rfloor (X)$ by the value given by $occ(a, X)$[1], whereas to monitor the amount of a outside the membrane we can observe the number of occurrences of $X \mid \left(b\right)^{L} \rfloor (Y)$ multiplied by $occ(a, X)$.

3.1.2 Faced Problems

The first problem we have faced is the definition of a compact runtime representation of terms and patterns of CLS. In fact the terms given by the CLS syntax grammar (see Section 2.1.1) have a huge number of nodes and this influences the performance of the stochastic simulation algorithm, that, as we see next, must walks each node in the subject tree. Moreover the CLS trees, as defined by the CLS syntax, are subordinates to a complex structural congruence that is expensive to verify at runtime. Thus we use a form of compressed representation of these trees that, in addition to reduce the number of nodes, allows to test the structural equivalence in a simpler and less expensive way.

Since the simulator must be able of doing real–time simulations involving millions of iterations, ours main problem is to obtain good performance of the computation step. Because the most expensive phase of each simulation step requires to find the set of all possible reactions in the system, we need a clever algorithm for doing that task. The naive algorithm, that try to unificate each pattern in each position of the subject tree, it is not usable because requires an exponential computational cost. In fact it demands to scan repeatedly the entire subject tree at each step.

In order to minimize the number of times in which the subject tree must be walked, we have developed a bottom–up pattern matching algorithm, inspired by the approach proposed by Hoffmann and O'Donnel [50], that, thanks to some data structures produced in a pre–processing phase, succeeds to find all the possible reactions in the term state doing a single walk of the subject tree. Moreover this algorithm allows to preserve great part of the computation between a step and the next one, recomputing the informations about matching only for the part of the state that is changed by the execution of a reaction.

In summary the idea behind the algorithm is to expand the term data structure in such way that each node is enriched with some informations about each pattern and parts of pattern that matches at this point in the subject tree. In order to

[1] Where *occ* is the function defined in Example 2.10.

doing this, the algorithm starts from the leaves and proceeds in bottom–up way. The matches of the leaves are computed by a *nondeterministic finite automaton*, whereas the way of inferring the matches of a node from the matches of the children are defined by a data structure (a sort of *transition table*) built pre–processing the set of rules.

Since to count the number of possible reaction require to do repeatedly some expensive combinatorics computations, we have enriched the algorithm with some level of caching of these computations.

Finally we have expanded the standard Gillespie's SSA in order to deal directly with rule schemata instead of ground rules.

3.2 Design

Existent Solutions and Similar Products

Currently does not exists any simulator for CLS (except an early C++ prototype simulator implementing a naive pattern matching algorithm with very poor performance).

A variety of software packages specifically developed both for *deterministic* or *stochastic* simulations are available. Among the deterministic ones we can mention software such as DBsolve [48], GEPASI [74], KINSIM [31], MIST [40], KINSOLVER [12], PLAS [102], ECELL [100] and Cellware [35]. Among the stochastic ones we find for example Stochsim [42], CytoSim and PSym [5], ECell [99], MCell [4], CellIllustrator [1], VirtualCell [8], SBW [6] and more can found for example in [7].

An interesting stochastic simulation software is SPiM : Stochastic π-calculus machine [81]. SPiM allow to simulate reactions expressed in pi-calculus, in real–time. Even if it is bases on the π-calculus [78, 89], a formal language for concurrent computational processes, whereas CLS is based on term rewriting, the SPiM simulator has been taken as an example of the features that the SCLS simulator must have.

3.2.1 Choices

Programming Language

For the development of the simulator we have chosen the F# programming language [96].

F#, a research project from Microsoft Research, is a multi-paradigm .NET language explicitly designed to be an ML (see next paragraph) suited to the .NET framework. It is rooted in the Core ML design, and in particular has a core language largely compatible with that of OCaml. Although it is a research project, still under continuous development[2], it has a product quality performance (see [97]). It merges seamlessly the object oriented, the functional and the imperative programming paradigms, allowing to exploit the performance, portability and tools of the .NET framework, without renounce to the benefits of functional programming.

[2]When this thesis is started the F# compiler was at the 1.1.13.8 release; in the relative small period of this thesis work it has reaches the 1.9.2.9 version with a great number of new features and improvements.

The first publications that introduce F# is [94] and at the moment two books are in publication [95, 83].

Functional Programming (FP) is the oldest of the three major programming paradigms[3].

Pure functional programming views all programs as collections of functions that accept arguments and return values. Unlike imperative and object–oriented programming, it does not allows side effects and uses recursion instead of loops for iteration. The functions in a functional program are very much like mathematical functions because they do not change the state of the program. In other words, once a value is assigned to an identifier, it never changes, so functions do not alter parameter values, and the results that functions return are completely new values. In typical underlying implementations, once a value is assigned to an area in memory, it does not change. To create results, functions copy values and then change the copies, leaving the original values free to be used by other functions and eventually to be thrown away by *garbage collector* (GC) when no longer needed.

In [36] Edsger W. Dijkstra pointed out that too many programmers rely on executing a program in order to understand it. The reason is that imperative programs lack of sufficient underlying formalisms to make guarantees about any of the most trivial of programs. As much the debugger is an useful tool, it is disheartening to need to use it to understand code. In contrast of this functional programs have solid theoretical foundations and therefore can more easily make runtime guarantees. In fact, tanks to the theory on which FP is founded, it is possible to prove various runtime properties of programs. While most programmers would not take the time to write proofs like Euclid's about their programs, it is compelling to think about creating sections of a program which are written so well that their properties are provable and understandable (an example is well show in [82]).

One of functional languages with greater success is ML (stands for *meta language*) is a general-purpose functional programming language developed by Robin Milner et al. in the late 1970s at the University of Edinburgh. The use of ML, thanks to the high level of abstraction, permits to reduce the difference between the specification phase and the coding phase, writing code more quick and leading to code that is more compact and contains fewer errors than the equivalent imperative code. Moreover ML allow the use of pattern matching that is an useful tool when we deals with symbolic computations.

In general terms programming functionally leads to more modular, generic, expression-oriented, and conceptually simple code. For a survey on benefits of functional programming see [15, 52].

[3] The first FP language, IPL, was invented in 1955, about a year before Fortran. The second, Lisp, was invented in 1958, a year before Cobol. Both Fortran and Cobol are imperative (or procedural) languages, and their immediate success in scientific and business computing made imperative programming the dominant paradigm for more than 30 years. The rise of the object-oriented (OO) paradigm in the 1970s and the gradual maturing of OO languages ever since have made OO programming the most popular paradigm today.

F# is a .NET language modeled on Objective Caml (OCaml)[4], an object oriented extension of ML. It was invented by Don Syme and is now the product of a team at Microsoft Research (MSR) in Cambridge, England.

Functional programming is the best approach to solving many thorny computing problems, but pure FP is not suitable for general-purpose programming. So, FP languages have gradually embraced aspects of the imperative and Object Oriented paradigms, remaining true to the FP paradigm but incorporating features needed to easily write any kind of program. F# is a natural successor on this path. With F#, it is possible to choose whichever paradigm works best to solve problems in the most effective way. It is possible to do pure FP (using a lot less mutable local state and making code more clear), but it is also possible to easily combined functional, imperative, and object-oriented styles in the same program exploiting the strengths of each paradigm. Being modeled on Objective Caml (OCaml), like other typed functional languages, F# is strongly typed but also uses inferred typing, so programmers do not need to spend time explicitly specifying types unless an ambiguity exists.

Some of the most popular functional languages, including OCaml, Haskell, Lisp, and Scheme, have traditionally been implemented using custom runtimes, which leads to problems such as lack of interoperability. F# is a general-purpose programming language for .NET, a general-purpose runtime. Further, F# seamlessly integrates with the .NET Framework base class library and then tweaked and extended to mesh well technically and philosophically with .NET. It fully embraces .NET and enables users to do everything that .NET allows. The F# compiler can compile for all implementations of the Common Language Infrastructure (CLI), it supports .NET generics, and it even provides for inline Intermediate Language (IL) code. The F# compiler not only produces executables for any CLI but can also run on any environment that has a CLI, which means F# is not limited to Windows operating system but can run also on Linux, Apple Mac OS X, and OpenBSD.

Linguistically, F# includes:

- The standard constructs of Core ML, essentially as implemented by OCaml;
- Type inference based on an instantiation of HM(X) with subtype and operator overloading constraints [79];
- A .NET-style nominal object model including classes, single inheritance, object expressions, properties and nominal interfaces associated with object values;
- A dot notation, where overloading is resolved in a type-directed fashion based on the type information available on a left- outside-in analysis of a file.

Like other .NET languages, F# derives much of its power from its reliance on .NET:

- Garbage collection;
- JIT and install-time compilation;

[4]F# has the same core language of OCaml: *CoreML*.

- Cross-language, inter operable generics, with automatic generalization (see Section 3.3.2).
- Relatively high performance, especially for floating point;
- Concurrent GC and SMP support;
- A vast array of high-quality libraries, including Windows Forms and Managed DirectX;
- Debuggers, CPU profilers and memory profilers;
- Portability across any Common Language Infrastructure (CLI) [54, 93] implementation, like Microsofts .NET Framework [76], Mono [106] and DotGNU [47]

F# embraces interoperability with CLI paradigms, for example:

- F# types and code can be used directly from other CLI languages;
- F# both generates and consumes generic CLI code; for example ML polymorphism is compiled as CLI generics, and generic definitions from other CLI languages can be used as polymorphic definitions to F# code;
- A simple, direct model of compilation is used, and optimization settings do not change the binary interface of F# components;

Graphical User Interface Library As regard the graphical user interface (GUI) of the simulator we have chosen to use the *Windows Forms 2.0* library (WF). This choice is motivated by the need of portability of the simulator on Mono (and thus on unix machines). There are a lot of possibility in the choice of GUI library for .NET runtime, and the most powerful and promising is *Window Presentation Foundation*. Unfortunately at the time of writing of this thesis Mono do not have yet support for WPF and therefore we have chosen WF because it seems to offer the best trade–off between power, ease of development and portability, allowing moreover an easy upgrade toward WPF (every element of WF has a correspondent in WPF).

The GUI is implemented by using a multi thread asynchronous model: the worker (= the simulator) runs on a thread at full regime sending asynchronous messages to a queue in the thread of the client (= the GUI). (The resulting interface is show in Figure A.1 on pag. 137.)

Steps in the Development Process

The development of stochastic simulators is error prone and introduces a lot of difficulties in the debug and in the proof of correctness. In fact the debug is made difficult by the stochasticity of the computations: different executions of the same simulation usually give different results. Moreover, the computation is driven from complex data structures of great dimensions, much difficult to trace during the execution of the program. Thus, we have divided the development process in three steps and we have tested and debugged the software in each step. The three steps are:

1. Data structures with SCLS semantics.
2. Simulation with naive algorithm, without variables support.
3. Simulation with pre–processing and support for typed variables.

3.2.2 Architecture

Logical Architecture of the Simulation

The logical architecture of the simulator, using Gillespie's direct method, is the same of any stochastic simulator that use the Monte Carlo's method (see Figure 3.1). The interpreter works in cycle: it repeatedly selects randomly a reaction among the possible reactions. The probability of the selection is determined from the concentration of reactants available and from the relative probabilities of every reaction. The reactions and the relative probabilities will be stored in a data structure that the code will visit repeatedly. The structure of this iterative process is not very different from an interpreter.

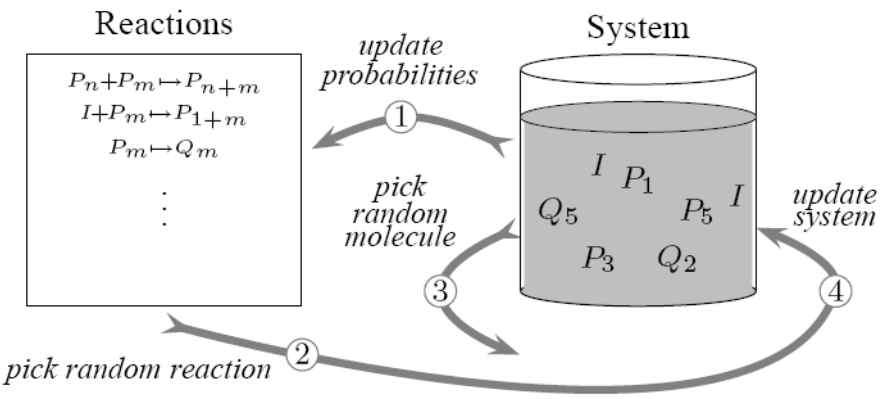


Figure 3.1: Logical architecture of a Monte Carlo simulator.

Namely, like all the Monte Carlo simulation methods (see Figure 3.1), a single step of computation of the main engine of the simulator consists of the follows activities:

1. to determine the probability of every reaction;
2. to choose randomly a reaction in agreement with its probability;
3. to choose randomly the molecules to which applying the reaction (in the set of possible candidates for the selected reaction);
4. to do the reaction and to update the state.

In practice, because the probability of a reaction is given by the number of possible ways of applying the correspondent rewrite rule, the first step look for all

the possible matches of the left hand side of each rule against the system state, that is represented by current term. This search must be exact in order to do the next reaction according to the correct probability distribution, thus techniques based on approximation or pruning are not usable. This is clearly the most expensive step of the algorithm.

Analogies with other Systems

This architecture is very similar to that of production systems (see Section 1.5). In fact in the simulator we can easy identify the main components of the architecture of a production system: the set of rules in a SCLS model acts like *production memory*, the CLS term representing the state of simulation can be seen as *working memory* and the Gillespie's procedure works as *conflict resolution strategy*.

Although, peculiar characteristics of SCLS are that the production memory is static (never altered during the simulation), and that the rules works on a working memory structured as tree. Moreover the Gillespie algorithm would require a stochastic conflict resolution strategy, that can not be found in productions systems.

In production systems, to collect production rules with matched conditions, is used an optimized pattern matching algorithm in which rules are pre–processed and compiled into a network of inter–related conditions. This is illustrated by the RETE algorithm designed by Charles L. Forgy in [43] and developed in [38].

The RETE algorithm is intended to improve the speed of forward–chained rule systems by limiting the effort required to recompute the conflict set after a rule is fired. Its drawback is that it has high memory space requirements. It takes advantage of two empirical observations:

- *Temporal Redundancy (state–saving):* the firing of a rule usually change only few facts, and only a few rules are affected by each of those changes.
- *Structural Similarity (node–sharing):* the same pattern often appears in the left–hand side of more than one rule.

The RETE algorithm uses a rooted directed acyclic graph where the nodes, with the exception of the root, represent patterns, and paths from the root to the leaves represent left–hand sides of rules. At each node is stored information about the facts satisfied by the patterns of the nodes in the paths from the root up and including this node. This information is a relation representing the possible values of the variables occurring in the patterns in the path. The RETE algorithm keeps up to date the informations associated with nodes in the graph.

When a fact is added or removed from working memory, a token representing that fact is entered at the root of the graph and propagated to its leaves modifying as appropriate the information associated with the nodes. When a fact is modified this is expressed as a deletion of the old fact and the addition of new fact.

In other words RETE is implemented building a network of nodes everyone of which represents one or more test found in the left hand side of a rule; the facts that are added or removed from working memory are processed by this net of nodes. At the base of the net are nodes representatives to an entire rule. When a fact arrives

to the base of the net, it means that all the tests of the left part of some rule are satisfied and thus that rule comes activated and inserted, with the record containing all the instantiations of the variable, in the set of satisfied rules (= conflict set). At the successive step we preserve the matches already computed and only new facts are tested against any rule left–hand side.

Architecture of the Developed Simulator To take advance of the fact that the set of rule is static we pre–process this set building some data structures that allow to reduce the computational effort needed for each simulation step at runtime.

This make favorable the architecture show in Figure 3.2.

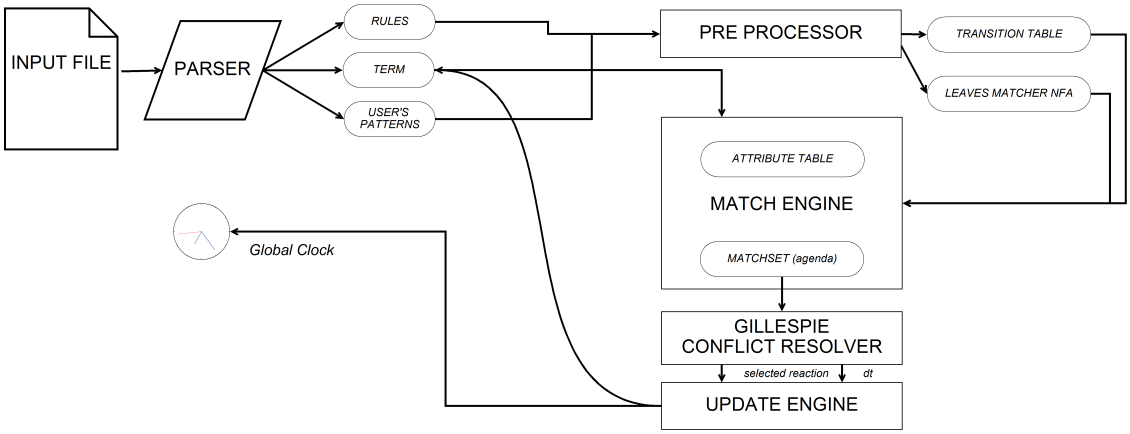


Figure 3.2: View of the architecture of the developed simulator. The procedures are indicates with rectangles whereas the data structures are indicates with ovals.

This architecture are composed by four logic components: a *preprocessing* module, a *match* engine, a *conflict resolution* module and an *update* engine.

- The *preprocessor* creates the data structures used by the match engine, from rules given as input: they are a sort of *transition table* and a *non deterministic finite state automaton* (NFA). The transition table, given a node, tells what parts of patterns are matched by the node according to its type and to the parts of patterns matched by the children. The NFA is the responsible of unification of sequences (and looping sequences) in the state term against sequence patterns in the rules.

- The *match engine* is responsible to find all matches of rule the schemata inside the state term. In summary they will walks and enrich with some attributes the tree of state term, starting from the leaves to the root. When the engine meet a leaf (that is a sequence) it asks to the NFA the list of the possible sequence patterns that can be unified with the leaf, together with the list of all possible bindings. Going back through the nodes it infers the attributes of a node according to the attributes of children and to the transition table, that was build in the preprocessing phase.

- The *conflict resolver* selects what rule to apply, among the set of possible reactions, according to their probabilities. The reaction to fire is selected by an extended Gillespie's algorithm that deals directly with rule schemata instead of single rules. Moreover this module returns the time elapsed by the selected reaction.

- The *update engine* at each step asks the match set of rule schemata to the match engine, get the selected reaction from conflict resolution engine and thus update the term state, incrementing the global clock by the time given time. This cycle will continuous until, or the end time of the simulation is reached, or a simulation step give a $\triangle t$ of 0. In the latter case the simulation is stopped as there are not any possible reactions and us they will not be any more.

3.3 Implementation

3.3.1 Data Structures

Because CLS is a term rewriting system, we have to deal with trees. More precisely with the abstract syntax trees of CLS terms and patterns (see Section 2.1). terms and patterns of CLS are defined by their abstract syntax trees, build of binary operators with a particular structural congruence. These trees are not suitable for runtime representation and a more clever representation can be used. In fact these syntax trees can be compressed finding a common representation of each structural equivalent class of terms and grouping together subtrees that are structural equivalent. That is: instead of doing trees made by the binary operators of CLS, that must be considered equal to a certain number of structural congruent trees, we can use trees made of an n–ary unordered parallel composition operator (corresponding to $\_ \,|\, \_$) and of a binary ordered operator corresponding to loop and containment operator $(\!(\_)\!)^L \rfloor \_$). This kind of trees have as leaves sequences composed by the an n–ary ordered sequencing (corresponding to $\_ \cdot \_$) operator. In this way we can exploit the unordered nature of the parallel composition operator ($\_ \,|\, \_$) grouping together subtrees that are structural equivalent, saving a great number of nodes and simplifying the implementation of structural congruence: two term are structural equivalent terms if have exactly the same representation. The reduction in the number of the nodes is a great advantage also for the pattern–matching algorithm that, as we see next, must walk the entire term at each step of computation.

Example 3.2. The SCLS pattern

$$a \cdot b \,|\, a \cdot b \,|\, c \cdot d \,|\, \big(a \cdot \widetilde{x} \cdot b\big)^L \rfloor \big(\big(a\big)^L \rfloor (X \,|\, a) \,|\, \big(a\big)^L \rfloor (a \,|\, X) \,|\, b)$$

that has the abstract syntax tree shown in Figure 3.3.a, are translated in the tree shown in Figure 3.3.b correspondent to

$$(a \cdot b) \times 2 \,|\, (c \cdot d) \,|\, \big(a \cdot \widetilde{x} \cdot b\big)^L \rfloor \big(\big(\big(a\big)^L \rfloor (X \,|\, a)\big) \times 2 \,|\, b)$$

(where $n \times T$ stands for a parallel composition $T \,|\, \ldots \,|\, T$ of length n).

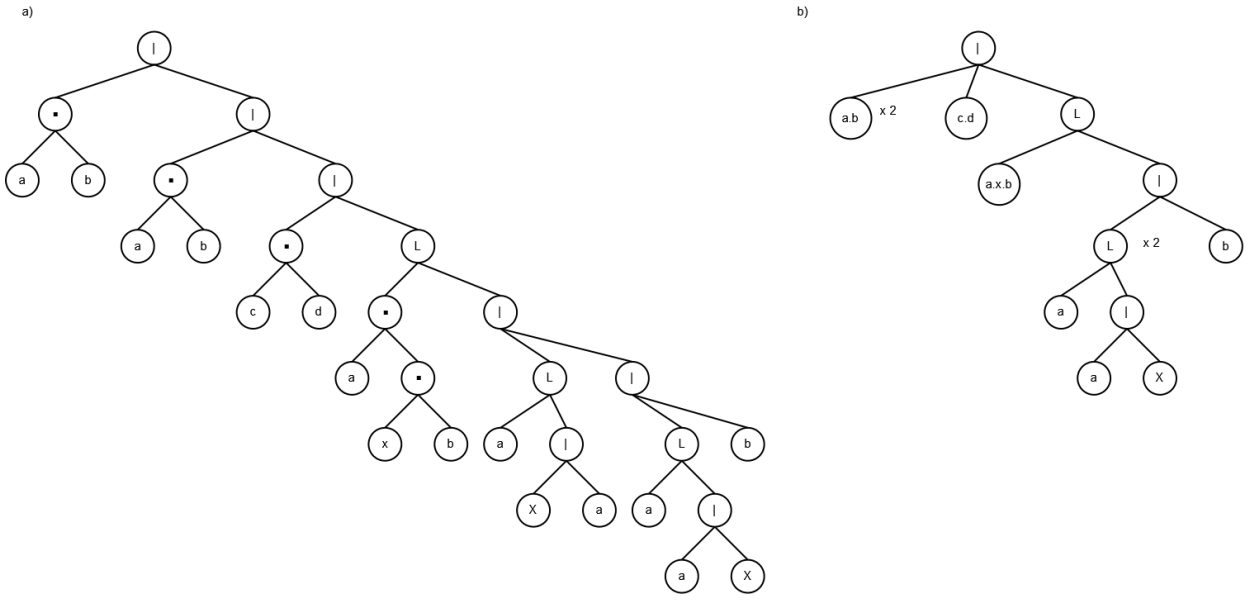


Figure 3.3: Comparison between the abstract syntax tree for the pattern and of the optimized tree of the CLS pattern $a \cdot b \,|\, a \cdot b \,|\, c \cdot d \,|\, \big(a \cdot \widetilde{x} \cdot b\big)^{L} \rfloor \,(\big(a\big)^{L} \rfloor \,(X \,|\, a) \,|\, \big(a\big)^{L} \rfloor \,(a \,|\, X) \,|\, b)$.

In Figure 3.4 is represented how the operators given by the syntax grammar of CLS terms and patterns are mapped into classes. The internal nodes of our trees trees are made of n-ary parallel composition operator (namely Compartment, instead of the binary $\_ \,|\, \_$) and binary looping and containment operator (namely Loop, corresponding to $\big(\_\big)^{L} \rfloor \,\_$) represented by classes derived from the Node abstract class. On the leaves we find Sequences (and LoopingSequences), representing terms made by the n-ary sequencing operator (namely Sequence, instead of the binary $\_ \cdot \_$), made by instances of classes derived form the Element class (ConstantElement, ElementVariable of SequenceVariable). Now we give a brief description of each category and of the data structures necessary for the simulation of CLS.

Element is an abstract class representing the components used to build the sequences, that are the leaves of our trees. From Element abstract class are derived the classes of *ConstantElement*, *ElementVariable* and *SequenceVariable*. The difference among these is that although the first represents a single elements of $\mathcal{E}$, the others represent a variable. The difference between element variables and sequence variables is that whereas the manner can be instantiated only with a single constant element, the latter can be instantiated with zero, one or more constant elements.

Node is an abstract class representing the nodes of our trees. From the Node abstract class are derived *Sequence* and *LoopingSequence*, *Compartment* and *Loop*.

The first two represent sequences made by the sequencing operator of CLS

$(\_ \cdot \_)$ and are implemented through an ordered list of elements. The difference between them is that the *LoopingSequence* can rotate whereas the *Sequence* can not. To efficiently compare different LoopingSequence we have to define some kind of *normal form* of these; we have chosen to store a LoopingSequence as the minimum of all the possible rotations in the lexicographical order.

A *Loop*, representing the looping and containment operator $(\!(\_)\!)^L \rfloor \_)$, is composed by one LoopingSequence, its membrane, and a Compartment, its content.

Compartments, representing the parallel composition operator of SCLS $(\_ \mid \_)$, are composed by a multiset of Node. Due to the unordered nature of compartment operator we can have advantage holding grouped sons that are equivalents according to the structural congruence of CLS. This is implemented through an hash table in which the key are computed on the nodes and have associated a number of repetition (a 64 bit integer value). In order to group together CLS terms that are equivalent according to the structural congruence of CLS we have defined an appropriate procedure to compute the hash key of a term. This procedure respects the order of elements into the sequences, not grouping together sequences with the same set of elements in different order, whereas groups together compartments with the same multiset of nodes, even if in different order. In other words the hash function is commutative for Compartments although is not commutative for Sequences and Loops. For more details relatively to the implementation of the compartment as hash table, and to the problems introduced by having to maintain consistent the chain of hash, see next section.

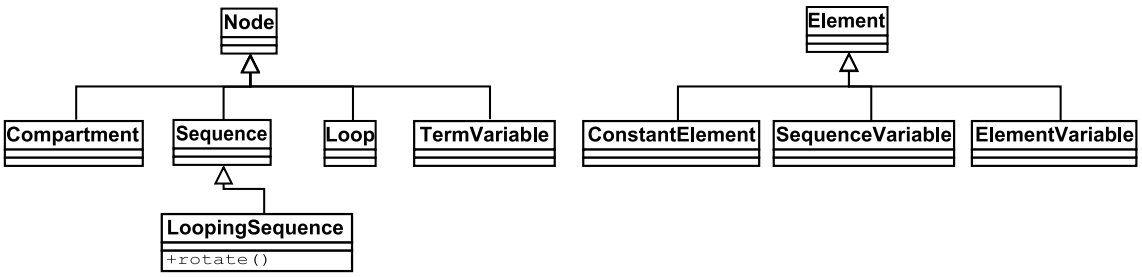


Figure 3.4: View of inheritance of data structures

So, in order to describe an model of SCLS (see Definition 2.7) it is necessary to define the following data structures:

Rule A rule of CLS is implemented by a class that is characterized by a name, a left hand side and a right hand side (that are CLS patterns) and by a rate function that, given a instantiation of the variables on the rule (a binding), return a float value (see Section 3.3.4). Moreover we store some additional informations, as for example if the rule is voidable; a rule is voidable if its left

hand side can be instantiated to ε. In case of voidable rule we must exclude empty bindings from its matches because of the $T\sigma \not\equiv \varepsilon$ condition imposed by CLS semantics (see Definition 2.6).

Model A model of CLS is implemented by a class containing a set of rules and a term representing the state of the simulation.

The Model class is enriched by some additional information. As example we store a data structure that manages the translation of string identifier in integer identifier. In fact, given an instance of the simulation problem (term + rules), the set of identifier is fixed (do not grows during computation); this allows to map the set of string identifier in a set of integer identifier that are more efficient to handle. Moreover we store in instances of Model informations about the patterns that the user want to monitor; although this problem is not closely related to the simulation problem we need to store these patterns because we delegate the count of such patterns to the pattern matching algorithm.

Moreover, in order to handle instantiations of variables we have defined the following structures

Binding A Binding represents an instantiation that map variable identifiers to appropriate nodes in which they are instantiated.

Match A Match represents the concept of context as defined in Definition 2.5; that is an occurrence of a rule left hand side pattern in the term. Thus a Match is composed by a rule identifier, a reference to some node inside of the term and the list of bindings that concur the match of the pattern in the specific location in the term. In the implementation we have chosen to group together different instantiations of variable of a rule, representing a single occurrence of the rule pattern, in a single match (see Section 3.3.2).

Implementation of Compartments as Hash Tables As we already seen, we have implemented compartments with hash tables. An hash table is a data structure often used to implement associative arrays, sets and caches. It can uses any data type as index, supports efficient addition of new entries, and the time spent searching for the required data is independent from the number of items stored ($\bigcirc$(k)). However, in the very rare worst-case, the lookup time can be $\bigcirc$(n). It works by transforming the *key* using a *hash function* into a *hash codes*, a number that is used as index of an array to locate the desired location (*bucket*) where the values should be.

Therefore the first step to use an hash table is to define the *hash function*: this function transforms the data type used by index in an integer that is used as index of an array. Ideally, different keys should be always translated into different indexes, but practically, is often used a data type that is greater of the integer data type that we use as index; thus the perfect hash function can not exist if we use certain data types as key. This is our case, in fact we try to compress in a space given by a 32 bit integer values a pairs made by a 32 bit integer value (the hash code the child) and a 64 bit integer value (the number of repetitions of the child). If two keys hash to the same index, because the correspondent records cannot be stored

in the same location, we must use one *collision resolution techniques*, among which the most popular and simplest are chaining. In the chained hash table technique, each slot in the array references to a linked list of inserted records that collide to the same slot. The insertion requires to find the correct slot, and to append the value to the end of the list in that slot; the deletion requires to search the list and to do removal.

The use of hash table to implement the compartment has the advantage of being able to group structurally equivalent (as $\equiv$ in Definition 2.2) terms in constant time; in fact this avoids to compare each son with each other. The way of doing this in F# is to implements the *IStructuralHash*[5] interface using a

$$Dictionary < \_ > (HashIdentity.Structural)$$

The use of hash table to implement trees presents also some disvantages. In fact a good hash function must return exactly the same value regardless of any changes that are made to the object. But this requirement is not respected in the case of naive implementation of trees with hash table. In fact, if we implement the compartments with hash table we have to deal with the following inconvenient: given a compartment, if a child, or one of the descendants subtree, is modified, we have that their hash code changes, leading to a inconsistent state. The compartment that contains this son will be stored, modified, in a bucket assigned on the base of hash code that has been calculated before the modification. In practice this causes that the son will not more accessible: if we try to access the value associated with the modified child, we will compute its hash code, that will be different to that one with the child was added (when it was not yet modified).

In order to avoid this situation it is necessary that each modification at a node is executed with a procedure that takes care to maintain consistent the overhanging chain of the hash codes (see Figure 3.5 and Listing 3.1). In particular when we modify a node is necessary, before removing it, to remove its parent to its grandfather (the parent of parent), continuing so until the root of the term is caught up (Figure 3.5.a). Only at this time it is possible to modify the node locally (Figure 3.5.b). Finally it is necessary to add the node (modified) to its father, its father to its grandfather and so on till the root (Figure 3.5.c).

Although the cost of this update procedure, if it is worth the assumption that the number of accesses to a term is smaller of the number of modifications, continues to remain convenient to maintain an hash structure respect to a sequential access structure (based on exhaustive comparison) like for example a list.

3.3.2 Search of Matches Algorithm

Because a SCLS rewrite rule is a triple (P_i, P_{ii}, f) that can be applied to a term T, if there exists a sub term of T that is structurally equivalent to a term that can be obtained from P_i through a valid instantiation of its variables, we have to deal with

[5] When we use a *Dictionary* $< \_ >()$ we use hashing on reference, considering different all the objects that are not the same instance; when we use it with *HashIdentity.Structural* we use the structural hashing defined implementing the *IStructuralHash*.

Listing 3.1: Procedure to keep consistent the hash chain in Compartments.

```
/// remove the path from this node to the root of term
member c.removeUp(parent_of_parent : Node option) =
    if parent_of_parent.IsSome
        then
         match parent_of_parent.Value with
                    | :? Compartment as cc -> cc.RemoveChild(c.parent.Value, 1L)
                    | _ -> (failwith "runtime error")
/// add the path from this node to the root of term
member c.addUp(parent_of_parent: Node option) =
    if parent_of_parent.IsSome
            then
                match parent_of_parent.Value with
                    | :? Compartment as cc ->  cc.AddChild(c.parent.Value, 1L)
                    | _ -> (failwith"runtime error")
/// add occurrence number of given child to children
member c.AddChild((node, occ):(Node*int64) ) : unit =
    let parent_of_parent =
                match c.parent with
                    | None -> None
                    | Some(p) ->  p.parent
    c.removeUp(parent_of_parent)
    match node with
        /// if we want to add a compartment then add its children directly
        | :? Compartment as comp ->
            for subChild in comp.children do
                c.AddChild(subChild.Key, subChild.Value * occ )
            done
        | _ ->
            /// check if the compartment already have this child
            let ok,value = c.children.TryGetValue(node)
            /// if so then only increments the number of repetitions
            if ok
                then  ( c.children.[node]<- (value + occ) )
                /// else add as fresh
                else  ( c.children.Add(node,occ) )
    node.parent <- Some(c :> Node)

/// remove a child from children
member c.RemoveChild(node, occ):(Node*int64) : unit =
    let parent_of_parent =
        match c.parent with
            | None -> None
            | Some(p) ->  p.parent
    c.removeUp(parent_of_parent)
    /// check if the compartment contains that child
    let ok,value = c.children.TryGetValue(child)
    if ok
        then
            /// if so, check the number of occourrence of that child
            /// if try to remove equals or more than how many child are present;
                then remove the child
            if(value <= times)
                                then    (c.children.Remove(child) |> ignore)
            /// else only decrement it occourrence's counter
            else    (c.children.[child] <- value- times)
                                c.addUp(parent_of_parent)
        ///else do nothing (there are nothing to remove)
```

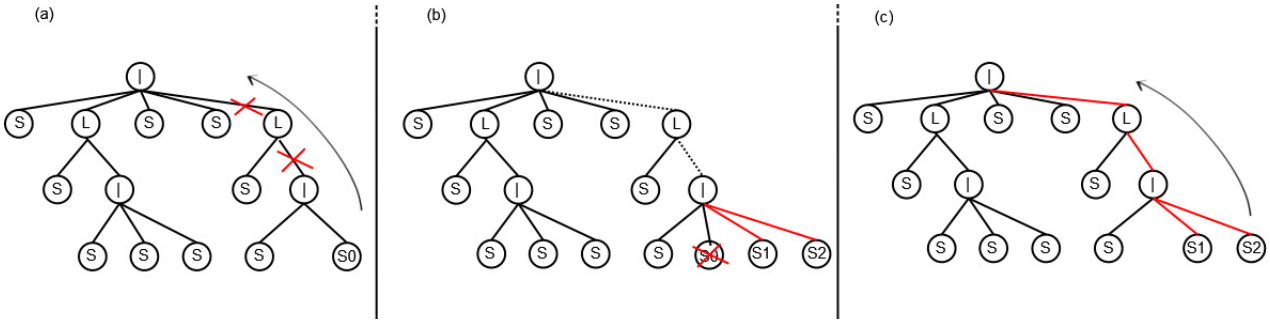


Figure 3.5: Example of update hash procedure; we remove $S0$ and add $S1$ and $S2$. We destroy the hash chain before the modification of the interested node (a). Then we local modify that node (b), and finally the hash chain is updated (c).

the following problem: given a pattern find all its occurrences in a term within the corresponding valid instantiations of the variables.

This is the most expensive task in each computation step, that is repeated millions of times during a simulation. Thus, an efficient realization of the search for matches phase is the main issue in the implementation of simulator.

The Problem of Subtree Pattern Matching

The problem of subtree pattern matching regards the detection of the occurrences of a *pattern tree* P of m vertexes as a subtree of a *subject tree* P of n vertexes, with $m \leq n$. Since the seminar paper of Hoffmann and O'Donnell [50], this problem has been studied by many authors, rendering the literature on this problem rather abundant with many definitions of the problem itself; a recent good reference is the book of Valiente [101].

We introduce the tree matching as follow :

Definition 3.3 (***Tree Matching Problem***)**.** A matching problem on trees consists of a finite set of *patterns* $p_1, \ldots, p_k$ in a *subject tree* t in S. A *solution* to a matching problem is a list of all the pairs (n, i, b), where n is a node, p_i matches at n with variable bound according to the binding b.

All method for tree pattern matching should be compared with the naive algorithm (based on a simple form of unification), which merely tries every pattern at every position in the subject tree.

The general notion of this problem on unordered tree has been shown to belong to NP-Complete class (for detail see [56, 90, 108]).

The subtree pattern matching can be classified according to the characteristics of input trees :

- we can have *ordered* or *unordered* trees; in first case two trees are the same only if they have the same children in the same order, whereas in the second case the order among children is not important;

- we have *k-ary tree with k bounded* if each node has a number of child that is in a set that is fixed; we have *k-ary trees with k unbounded* if the number of children (the number of operator with different arity) are not a priori bounded;
- we can have *rooted* or *unrooted* trees (tree with a root node or without, or with more than one root nodes);
- we can have *labeled* trees, if each node has associated an identifier, or with *unlabeled* trees;
- we can have to deal with the presence of *logical variable* in the pattern trees or we can have to find only *constant* trees;
- we can have to deal with different *notions of occurrence* (see next section).

Moreover the pattern matching algorithm can be used to find *exact* matches (also called *isomorphisms*) or *approximate* matches, that is to try all the subtrees that are similar enough to some patterns, according to some notion of similarity (for example a distance measure given by the sum of the costs of deletion, insertion and relabel operations on tree nodes necessary to obtain the term form the pattern).

This differentiation in the characteristics of the problem implies that does not exist an efficient and sufficiently general algorithm that succeeds in dealing with all these typologies of problems. Instead, there are a lot of papers that propose algorithms that solves some instances of the problem very efficiently, taking advantage of the assumptions on the input problem instances. For example we can mention [25, 67, 69, 70, 71].

Subtree Pattern Matching in CLS

The main difficulties that we find in classifying CLS in one of the classes proposed in literature are the following.

Arity of the Operators All works proposed in the literature analyze instances of the problem were, given an input, we have to deal with a finite alphabet of operator with fixed finite arity. In CLS pattern matching, instead, given an input problem we can have to deal with an infinite set of operators with finite arity. In fact, as we seen in the previous section, since we use a representation of the abstract syntax tree of terms in which a node corresponding to a parallel composition operator may have an arbitrary number of children, we have to consider such an operator not as a single binary operator but as a set of operator, each with different arity. Hence we have to deal with an infinite set of operator with finite arity as

$$\mathcal{E} = \{(\_)^L \rfloor\_, \mid_1, \mid_2, \mid_3, \ldots, \mid_i, \ldots\}$$

where with $\mid_n$ we mean a parallel operator with fixed arity of n.

Also when we know the instance of the problem to simulate we can not limit a priori the set of operators with different arity that we will need in order to make the simulation.

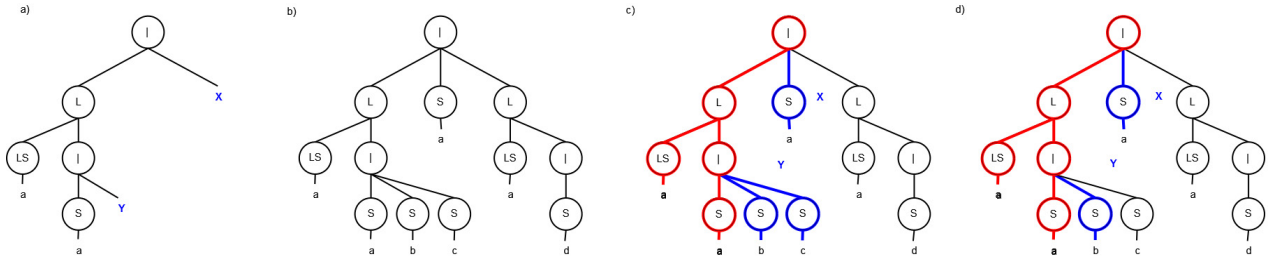


Figure 3.6: Example of notion of subtree occurrence required in CLS pattern matching. a) is the pattern tree, b) is the subject tree; in c) is shown a valid instantiation of the rules whereas the instantiation in d) is not. This because Y must be bound to all the children of the relative compartment that are not required to match a constant sequence.

Notion of Occurrences Most of papers that study the tree pattern matching problem propose algorithms that deals with a restricted notion of pattern occurrence in the subject tree: a pattern match on a node n in the subject tree if the root r of the pattern has the same label of n and there is a one-to-one map of each child of r to a child of n in the subject tree (there is not any child of n that is not mapped to some child of r) and so on recursively. In other words it is like to require that there exists a bijective function that map each child of the root of the pattern to each child of the node in which it match, and so on till the leaves. If a node p of the pattern match to a node n of the subject tree then all the children of n match with a child of p, thus there are not nodes in the subtree of n that are not involved in the match.

Unfortunately in the case of CLS pattern matching must deals with an hybrid notion of occurrence. Although for the looping and containment operator $(\!(\_)\!)^L \rfloor \_)$ we have to deal with the the notion of occurrence just defined, for the parallel composition operator $(\_ | \_)$ we have to deal with a relaxed notion of occurrence. A parallel composition pattern node p will match at a node n in the subject tree if each child of p match with a child of n, even if there are child of n that do not match any child of p. An example is show in Figure 3.6

Semi Ordered Trees For parallel composition operator $(\_ | \_)$ operator we have to deal with unordered trees, although for looping and containment $(\!(\_)\!)^L \rfloor \_)$ operator we have to deal with ordered trees[6].

In summary the instance of pattern matching problem that we have to consider is *exact pattern matching* on

- *semi ordered* (ordered for loop but unordered with compartment)
- *k-ary* with *k not bounded*

[6]The sequencing operator $(\_ \cdot \_)$ is here not considered because the sequences are the leaves of our trees.

- *rooted*
- *labeled*

trees, finding subtree occurrences with *typed variables*.

Moreover an important characteristic in the simulation of SCLS is that the set of pattern tree is static.

In order to take advantage of this we have developed a bottom–up, pre–processing based, algorithm, inspired by the idea of [50]. We introduce the algorithmic idea in the next section and then we look at how that idea is adapted to the case of CLS.

Our main goal is to develop an algorithm that has the same advantages of RETE [43, 39] (see Section 1.5) (*state–saving and node–sharing*) and that takes advantage of assumptions on CLS terms.

The Bottom–up Algorithm

To find all the possible matches of the left side of the rules, we have implemented an extension of the bottom–up algorithm of Hoffmann and O'Donnell [50] (which takes advantage itself of ideas of the algorithms for multi–pattern matching on string based on pre–processing [10, 62, 24]). The basic idea is to exploit that the set of patterns is static doing a pre–processing phase; in this way, at match time, we succeed in walk the tree of the term a single time, in order to find all possible matches (thus with linear cost in the number of nodes in the subject tree).

In [50] are proposed two algorithmic ideas for solving tree pattern matching. The bottom-up approach, that find matches traversing the tree from the leaves to the root, is a generalization of the Knuth–Morris–Pratt [62] string matching algorithm. The top-down approach reduces tree matching to string matching problem and it is based on creation of some automata, that allow to solve the problem starting from the root to the leaves.

Between the bottom–up and the top–down approach, proposed in [50], we have chosen the first because it is characterized by faster matching and better response to local changes, even if it has exponential preprocessing time. The top–down approach instead has better preprocessing times but worse update behavior. How an algorithm for pattern matching responses to local changes is very important, in fact in applications of tree replacement, as CLS (term rewriting), the same set of rules is used many times (as more than 10^6) and each replacement causes a local change in the subject tree. So our pattern–matching technique can spend much time in pre–processing of rules (being a fixed set the pre–processing phase is executed only one time) and should be able to respond incrementally to local changes in the subject avoiding to repeatedly scanning of the entire tree.

As presented in [50], the key idea of the bottom-up matching algorithm is to enumerate and then find all patterns and all parts of patterns which match at each point in the subject tree. We assume to have a fixed set $\mathcal{E}$ of operators b_i with fixed arity q_i. Let n be a node in the subject labeled with the *q-ary* symbol b, and suppose we wish to compute the set M of all those pattern subtrees, other than node variable (v), which match at n. (Since v matches anywhere, we always have a match of v.) Suppose we have already computed such sets for each of the sons of n,

and call these sets, from left to right, $M_1 \dots M_q$. Then M contains v plus exactly those pattern subtrees $b(t_1 \dots t_q)$ such that t, is in M, for $1 \leq i \leq q$. Therefore we could compute M by forming trees $b(t_1 \dots t_q)$ for all combinations $(t_1 \dots t_q)$, where the t_i are chosen from M_i, and then asking whether each candidate for membership in M is a sub pattern. Once we have assigned these sets to each node in the subject tree, we have essentially solved the matching problem, since each match is signaled by the presence of a complete pattern in some set.

Note that there can be only finitely many such sets M, because both the set of sub patterns and the set of operator are bounded. Thus we could *pre–compute* these sets, code them by some enumeration, and then construct tables. Given a node symbol b and the codes of the M_i, these tables give the code for M. In the case of a *q-ary* symbol b, we would have a *q-dimensional* matrix for that symbol.

Given such tables, the matching algorithm becomes trivial: it traverses the subject tree in postorder and assigns to each node n the code c representing the set of partial matches at n as discussed. The tables consist of arrays, one for each alphabet symbol. If a node n is labeled with the *q-ary* symbol b, the *q-dimensional* array for b is used. The code c at n is the value indexed by the tuple $(c_1 \dots c_q)$ where c_i, is the code assigned to the i-th son of n (from the left). If the set represented by c contains the i-th pattern, then the pair (n, i) is added to the solution. Indicating with s the size of subject tree and with m the number of matches founds, the matching time of this algorithm is clearly $\bigcirc(s)$ for computing all codes plus $\bigcirc(m)$ for listing the solution.

An example of application of this algorithm is shown in Figure 3.7 (from [50]).

There is some similarity between bottom-up matching and formal parsing methods such as LR(k) parsing. In both cases a finite number of possible configurations are precomputed, and tables are formed to drive the parsing/matching process. As with LR(k) parsing, our tables will sometimes be very large. When a local change is made to a subject tree, matching codes must be recomputed for the changed portion and some ancestors of the changed portion. In [50] is shown that the number of ancestors whose codes must be recomputed is bounded by the largest height of a pattern.

In summary the bottom–up algorithm consist in two phases :

- *Preprocessing time:* Scan the rules enumerating all pattern subtrees and building some transition matrices. For each operator with arity q we have a q–dimensional matrix that, given the codes of pattern subtrees of each child match, give the codes of pattern subtrees that match the node with which the operator is associated.

- *Match time:* Starting from the leaves, scan and enrich with attributes each node of the subject tree according to the type of the node and the relative transition matrix. Each node that has the code of the top–level of some pattern has a match of the corresponding rule.

a) $\mathcal{E} = \{ a(\_,\_) , b , c \}$
$p_1 = a(a(v, v), b)$ and $p_2 = a(b, v)$

b)
Set 1 = $\{v\}$,
Set 2 = $\{b, v\}$,
Set 3 = $\{a(v, v), v\}$,
Set 4 = $\{a(b, v), a(v, v), v\}$,
Set 5 = $\{a(a(v, v), b), a(v, v), v\}$.

c)
Table for node label *a*.

	Right son				
Left son	1	2	3	4	5
1	3	3	3	3	3
2	4	4	4	4	4
3	3	5	3	3	3
4	3	5	3	3	3
5	3	5	3	3	3

Table for node label *b* 2
Table for node label *c* 1

d)

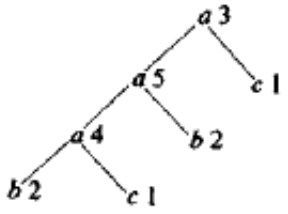


Figure 3.7: Example of Hoffmann and O'Donnel algorithm applied to the problem in which the patterns are shown in a); The alphabet $\mathcal{E}$ of operators is *a, b, c* , where *a* is binary and *b* and *c* are nullary symbols. In b) are shown the possible sets of pattern subtrees. Thus, assigning a 4 to some node *n* of a subject tree indicates that all the members of Set 4 matches at *n*. In particular, p_2 matches. Assigning 5 implies a match of p_1. c) are shown the tables for *a,b,* and *c*. For instance, the entry at (3, 2) in the table for *a* is 5, because at the left son we have a match of both *a(v, v)* and *v*, and at the right son of *b* and of *v*. For the nullary symbols *b* and *c* the tables are 0-dimensional, consisting of one entry each. d) shows the complete assignment of sub pattern codes when using the bottom–up algorithm with these tables. Note that p_1 matches at the node with code 5 and p_2 at the node with code 4.

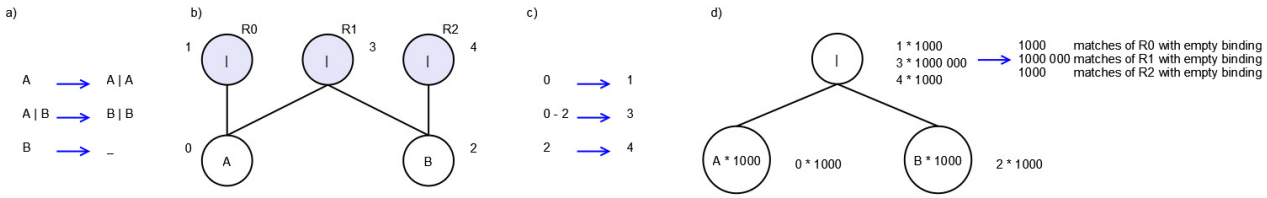


Figure 3.8: Example of pre–processing data structure generated for the Lotka simulation. In a) are shown the rewrite rules. In b) are shown the corresponding sub–patterns enumeration and in c) the corresponding transition table. In d) are shown the attributes computed for a term $A \times 1000 \,|\, B \times 1000$; the attributes of the root node are the codes corresponding to the full match of rule $R0$, $R1$ and $R2$.

The Bottom Up Algorithm Applied to CLS Pattern Matching

The main differences between proposed tree matching algorithm and the case of CLS are the following

- The children of CLS parallel operator are *not ordered*; therefore the transition matrices can't be used as they are defined.

- The *number of operators with different arity* (the cardinality of the operators alphabet), that are fixed in proposed algorithm, are *not bounded* a priori in CLS. (This causes that the set M described before is not bounded and thus that we can not reason on sets of parts of patterns, but on single parts of patterns.)

Therefore it is necessary to modify the notion of transition matrix so that the parallel operator requires the presence of an arbitrary number of sons which can be taken in any order. Thus the algorithm applied to CLS works in this way. In pre–processing phase it scan the set of rules building an enumeration of all the full patterns (entire left hand sides of rules) and part of patterns (subtrees of left hand side of rules) building a set of transitions; these transition, describing what parts of pattern are matched by a node according to its type and the set of parts of pattern matched by the children, are merged in a transition table. (For example for a compartment node of a pattern the transition table will ask the set of parts of patterns the must be matched by the children, in any order, to match this part of pattern.) Moreover in the pre–processing phase an NFA that match each leaves pattern are build. At runtime, at each step of simulation, the algorithm walk the subject tree and, using the NFA and the transition table, enriches each node with a set of attributes that tell what parts of patterns are matched by the node. The presence of the code of a sub pattern that is also the root of a left hand side of a rule means that the rule has a match on this node. In order to take account of variables we build the binding of variables of a match while we compute these attributes.

See Figure 3.8 for an example of application of the pre–processing pattern matching algorithm applied at the case of CLS.

We now look more in details at the necessary data structures and the procedures; these are all grouped in the *Engine* class.

Data Structures In order to implement the algorithm proposed by [50], we need to define the data structures representing a subtree of some pattern (that we call SubPattern), the data structure correspondent to the transition matrices of the Hoffmann's algorithm, (the TransitionTable), and the Attribute with which we enrich the the term tree.

Sub Pattern This data structure groups together all the information regarding a sub tree of the left hand side pattern of some rule. These are an integer identifier code, a reference to the node that is the root of the correspondent subtree, and a set of rule identifiers that indicates for which rules the subpattern is also a full pattern (that is if the root of the correspondent subtree is also the root of a left hand side of some rule). If this set is not empty, a node matches the left hand side of some rule.

Transitions and Transition Table In a Transition record we store all informations regarding one possible way of inferring an attribute for a node according from its type, the attributes of children and possibly required term variables. From the set of all Transitions is made a data structure corresponding to a the transition matrices entry of the Hoffmann's algorithm: the TransitionTable. It associates each operator with a set of possible transitions; these are made of an array of required children attributes (together with the number of times in which they are required), an array of variables to bind and a go–to entry; that is the identifier of the subpattern inferred if all the required conditions are satisfied. (See Figure 3.8.b)

Attribute In the Attribute data structure we store all informations that we need for enrich the term tree. Namely they are the follow :

- a subpattern identifier that indicates which subpattern match the node that is attributed;
- an array of references to the children involved by the transition that has created this attribute; this array will be useful when we are in front of the problem of removing only the attributes involved by a certain rule match;
- a list of bindings (in compressed way as see Section 3.3.2);
- an integer value representing the number of times in which the attributed node satisfies the subpattern indexed by subpattern identifier field;
- an array of variables bound by the transition that has generated this attribute;
- a boolean field that indicates if the transition that has generated this attributes has left not binded some children in the sub tree of the node. If so the attribute can not be used to inhering the match of some node of type looping and compartment. (As we have seen in Section 3.3.2 a match

of a part of pattern rooted in a node of type looping and containment requires that there are not nodes not bound in the subtree of the matched node in the subject tree.)

Procedures

Pre–processing Phase This phase is executed by the constructor of the Engine object.

First of all we merge the patterns of the left hand sides of the rules with the pattern that the user want to monitor. This is because, even if these patterns are not related with the simulation problem we must know also their matchset in order to plot their evolution.

Then we build the transition table for each operator (namely for Loop and Compartment). In order to do this we iterate through the array of patterns executing a procedure that traverse each note in the pattern tree. Given a node such a procedure generates the corresponding description of the sub pattern that the node represents, and, in the case in which the node is not a leaf, the corresponding transition table entry. (See Figure 3.8.b and 3.8.c) Each sub pattern is accumulated in a list through a procedure. This procedure adds the informations build for the subtree of the node to these already computed, according to some logic of sharing of sub patterns among different rules. The same logic of *node–sharing* is followed by the procedure that store of the transition table entries.

Match Phase We attribute the term tree, starting from the leaves (= sequences) using a non deterministic finite automaton (see Section 3.3.2); given such attributes on the leaves, we compute the attribute of each node in bottom–up way.

That computation is different according to the type of the node. For Loop nodes we look at all the possible transitions in the transition table for Loop operator; for each of these we look at the required attributes in the attributes list of the membrane and content, and, if we find these, we simply assign to the current node the attributes given by the goto field of the transition. For Compartments we act in a more complex way, in fact we have to find the required attributes for each transition among the attributes of children.

The complexity of this phase is high for two reason: the first is the unordered nature of the required attributes and of the children, and the second is the fact that a single required attribute (meant with a fixed number of times in which must be matched) can be satisfied by some different children and, symmetrically, that a single child (meant with a certain number of repetitions) can satisfy different required attributes. Moreover for each possible way of take the required attributes we have to bind the children that are not required to match some attribute to eventually required term variables. In the simplest implementation we store all these attribute in a separate table indexed by the node instance.

While we go back from the leaves to the root of term, when we infer an attribute corresponding to the top–level of a pattern we add a match, regarding the corresponding rule, to the matchset. (See Figure 3.8.d). In this way we walk the tree one time in order to know all the possible reactions.

Matching of Sequences by NFA

We have chosen that the leaves of our attribute tree are the Sequences, and thus we need an unification procedure that, in order to assign attribute to these leaves, compute the unification between Sequences and Sequence patterns.

In order to carry out this task we have implemented a *nondeterministic finite automaton* (NFA)[7]. An NFA is a finite state machine where for each pair of state and input symbol there may be several possible next states; it is non-deterministic in that, for any input symbol, its next state is not uniquely determined, but may be any one of several possible states. The implementation keeps a set data structure of all states in which the machine might currently be. On the consumption of the last input symbol, if one of these active states is a final state, the machine accepts the input.

We use NFA for doing unification. Namely, given a sequence, we want to know all the possible pairs (*pattern code* , *binding*) regarding all the sequence patterns that can be unified with that sequence. More in details we have to unify an ordered list of Elements with a set of ordered list of Elements, ElementsVariables and SequenceVariables; the Elements unify only with the same Element, the ElementVariable unify only with exactly one of any Element, the SequenceVariable unify with an ordered list of zero or more Elements.

Example 3.4. If we have the following sequence pattern within the left hand side of a rule

$$\text{a } \cdot\widetilde{x}\cdot\widetilde{y}\cdot z\cdot\widetilde{x}\cdot\widetilde{y}\cdot \text{ d } \cdot\widetilde{x}\cdot\widetilde{y}\cdot z\cdot\widetilde{x}\cdot\widetilde{y}\cdot \text{ a}$$

the solutions of unification, against the following ground sequence

$$a\cdot d\cdot a$$

are

$$\begin{gathered}\{(z=d,\widetilde{x}=\varepsilon,\widetilde{y}=d\cdot d\cdot d\cdot d\cdot d\cdot d),\\ (z=d,\widetilde{x}=d,\widetilde{y}=d\cdot d\cdot d),\\ (z=d,\widetilde{x}=d\cdot d,\widetilde{y}=d\cdot d),\\ (z=d,\widetilde{x}=d\cdot d\cdot d\cdot d,\widetilde{y}=d),\\ (z=d,\widetilde{x}=d\cdot d\cdot d\cdot d,\widetilde{y}=\varepsilon)\ \}\end{gathered}$$

[7] Nondeterministic finite automaton were introduced by Michael O. Rabin and Dana Scott in 1959.

The constructor of the NFA takes an array containing all the patterns of all the sequences that appear in the rules. Given such an array it generates an NFA for each sequence and then it merges them in a single automaton, sharing common states and transitions among different patterns.

As regard the variables, the NFA transitions are built in way that the bindings are created when the variable is met for the first time, and are stored within each active state. At the time in which the transition to match an already seen variable is created, it is built in way that the transition is executed only if the next n symbols in the input sequence are the same of the n symbols sequence that are bond to the variable, in the binding associated with the state from which the transition starts.

In other words, in the match phase, if we are on i–th position on input sequence and we have a transition that requires to match an already seen variable var, we do a *lookahead* of n symbols, where n are the number of elements of the sequence bound to the variable var in the binding associated with current state. If we succeed in comparing the lookahead with the list of elements bound to the variable then we activate the arriving state for the $i + n$ position on input sequence. In order to do this we use an array of sets of pairs ($active\,state$, $active\,binding$), one for each position in the input sequence.

The difference between ElementVariables and SequenceVariable is that the firsts are constrained to be bound to one and only one Element of input sequence, whereas the seconds can be skipped (bound to ε) and thus has a loop in the binding state that allows to bind more than one Element (see for example the transitions on states 2, 3, 4 and 5 in Figure 3.9 for details).

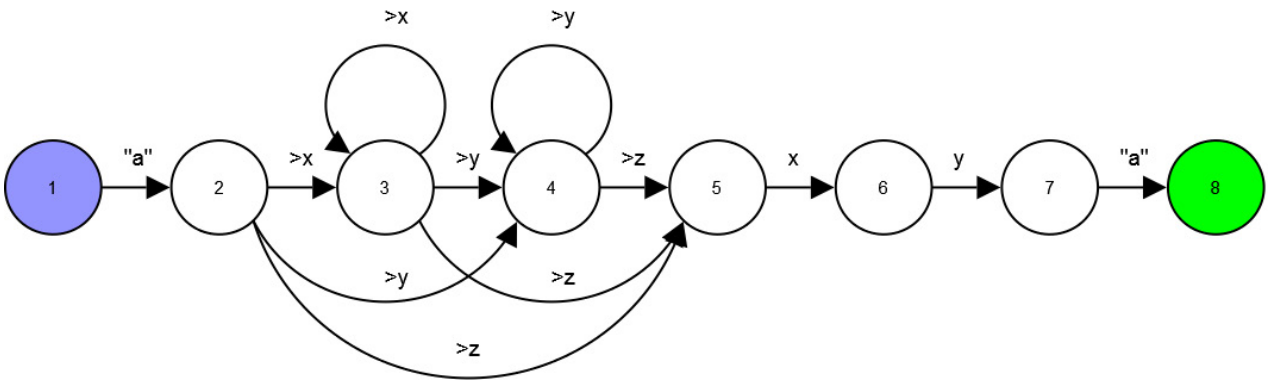


Figure 3.9: The NFA built for matching the a.$\widetilde{x}$.$\widetilde{y}$.z.$\widetilde{x}$.$\widetilde{y}$.a sequence pattern. With $>$ var on the transition arrows we mean that any read Element are bound to the variable identifier. With var we mean that the transition can be done if the next n symbols in the input sequence are the same of the sequence, of n symbols, bound to var int the binding associated with start state. Constant elements are indicated between quotes.

The NFA match phase is very similar to a standard driver of NFA: it starts on starting state, reads a symbol from the input sequence and makes all the possible transitions. They activate all the arrival states withing possibly created bindings. At the end of the input sequence, if there are some active states that are also final

states, it signals the matches of the relative sequence patterns with correspondent binding.

Regarding of the classic unification problem, we have in addition the problem of the management of the looping sequences; in fact they can be rotated, and, according to the definition of structural congruence of the CLS (see Definition 2.2), they must be considered equal to each of all their possible rotations. The solution is to test each rotation of a looping sequence against the NFA.

The NFA performance is enhanced by the use of *memoization pattern* like discussed in the next section.

The Term Updating Procedure

Even if the main goal of the developed pattern–matching algorithm is to allow the search of matches in the term with linear cost, it have also an important feature that allow to improve the efficiency of every step of the simulation. We can do like expert systems that takes advantage of *state–saving*, preserving, between an iteration and the next one, the informations that are not modified by an application of a rule. In fact, between an iteration of our simulator and the next one, we need to re–compute only the attributes of the modified part of the tree. More precisely if we modify a generic node n we need to re–compute the attributes regarding the modified children of n (at the worst case all the subtree rooted in n) and the attributes of the nodes on the path from root of tree to n (see Listing 3.2). This concurs to preserve all the remaining attributes from a iteration to the next, improving the performance in way depending on the complexity of the tree. The boost is quite oblivious but are not inferiorly limited and in the worst case, that is the case of each rule is applied to the root of the term involving all the children on the root, we do not have any speed up.

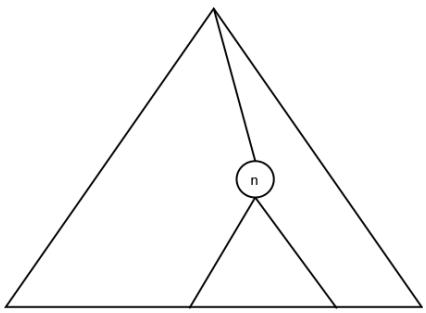


Figure 3.10: Local changes to subject tree after a rule application on node n. The attributes that need to be recomputed are these of n, of its subtrees and of the nodes on the path from n to the root of the term.

Further Optimizations

Here we present some further optimization ideas that improve the performance of the CLS pattern matching engine.

Listing 3.2: Procedure to update term attributes after a local changes. The procedure can be expanded to store the list of nodes with removed attributes, in order to remove also the matches related to touched nodes.

```
member x.update(m: Match) =

    let rec RemoveUp(n:Node) =
        if n.parent.IsSome
            then RemoveUp(n.parent.Value)
            x.table.Remove(n) |> ignore

    let rec RemoveDown(n:Node) =
        match n with
            | :? Compartment as comp ->
                    for keyVal in comp.children do
                        RemoveDown(keyVal.Key)
                        x.table.Remove(keyVal.Key) |> ignore
                    done
            | :? Loop as loop ->
                    RemoveDown(loop.content :> Node)
                    x.table.Remove(loop.membrane)|> ignore
                    x.table.Remove(loop)|> ignore
            | _->() /// we are on leafs
        x.table.Remove(n) |> ignore

    /// remove the attributes of the node
    x.table.Remove(m.where) |> ignore

    /// remove the attributes of the nodes on the path
    /// from the node to the root of the term
    RemoveUp(m.where)

    /// remove the attributes of the nodes on the subtrees
    /// of the involved children
    for child in m.involved_children do
        RemoveDown(child)
    done
```

Using Compressed Information About Term Variable Bindings To improve efficiency, it is useful to store the information about all possible bindings in a compressed way, expanding it only when need.

Given a transition that can give an attribute for a node n, it requires that some attributes are present among the attributes of some child of n. If the transition needs also some variables of type term, each of these can be bound to one of the power set[8] of the set of children that are not required to satisfy some required subtrees. The cardinality of the power set grows exponentially (for example with 10 sons the number of possible bindings is 2^{10}). Moreover a big part of these binding will be not valid to the upper level; a match of a loop and compartment operator does not allows that some node in the subtree of the matched node to be unbounded. Thus it is profit to store only the binding in which the term variable are bond to all the remaining children, and explode this information only when is required: namely when we have term variables in the top–level of a rule.

In other words we observe that any time that a term variable, in the bottom–up inference algorithm, pass up through a Loop, the term variable are always forced to be bound to the maximum part of remaining children. Thus, with the exception of the case in which a term variable is present at top–level of a rule, we can consider that a term variable is always bound to all the children not required to make a transition. (see Figure 3.6). When (and only when) we have a term variable at the top level of a rule we choose what binding has associated exploding the power set and randomly taking one of these.

Attribute Table as Direct Acyclic Graph We can do more node sharing compressing the attribute table tree in a *direct acyclic graph* form.

A direct acyclic graph (DAG) is a directed graph that not contains cycles. This means that if there is a route from node A to node B, there is no way back.

It is possible to group together the node in the subject tree in classes of structural equivalence taking advantage of a simple observation: each node in a class will have always the same attribute set. So, if two node are in the same class of structural equivalence, they have the same subtree and thus satisfy the same sub patterns and have the same set of attributes. This means that various instances of sub patterns within the subject tree have the same set of attributes. Thus we can share that list, obtaining *node–sharing* in the subject tree.

This optimization leads to use less memory and also to faster access to match set, even if it requires a more complex management of the addition and remotion of the attributes based on a reference counter.

Memoization pattern

Cause of the high quantity of computation repeated from a step to another, we have made extensive use of *memoization pattern*. It is a form of computation caching that allow to compute the results only at the first time we meet an instance of input; at the successive times we avoid the computation returning the cached result. Here is an example

[8] Given a set S its power set is the set of all subsets of S.

Listing 3.3: *Memoization* pattern example.

```
let memoize f =
    let cache = Dictionary<_, _>()
    fun x ->
        let ok,res = cache.TryGetValue(x)
        if ok then res
        else let res = f x
             cache.[x] <- res
             res
```

In the previous example the generic type of `memoize` is automatically computed: `('a -> 'b) -> ('a -> 'b)`. This makes the code short and clear. This is known as *automatic generalization*, and is a key part of type inference and succinct coding in all typed functional languages.

For preventing that the size of the cache can grow up unbounded (causing memory overflow) it is possible to add a private size limit field. When try to add a new element, first check the actual size of the cache; if it is lower than size limit we can remove an element from the cache according some policies (for example *least recently used*). If the cost of the computation of the results that we cache are significant, we still have a performance gain.

This kind of caching is used for example in the expensive computation of factorials: given a model, therefore a fixed set of rule, we have to compute at each iteration the factorials of each number of repetitions of each reactants of each rule. Because the rule set are fixed, at each step are required the same factorials independently from the reagent amount that is found in the current state.

This kind of caching is also used by the computation of attributes of leaves, by the NFA. Because we will test some Sequences repeatedly, we cache these results retrieving according to the structural congruence of sequences.

Alternatives

We now examine some alternatives for implementing pattern matching for CLS.

Using ML pattern matching An examined option would been to compile the CLS pattern matching in ML pattern matching. Unfortunately the following differences between the two notions of patterns make this approach unfeasible.

- The patterns of ML are composed of ordered operators; pattern of CLS are composed by the compartment operator whose sons are not ordered (see Section 3.3.2).
- The patterns of ML are composed of a finite set of operator of finite arity; pattern the CLS are instead composed from operators with an unbounded number of children.

Using Expert Systems Shell It would also been possible to use the inference engine of some existent production systems implementing RETE like as example

CLIPS [2]. We would generate CLIPS code from CLS input file and then run it by CLIPS engine. If fact it is possible to express CLS terms like CLIPS fact and CLS rule (on terms) as CLIPS rule (on facts). See Listing 3.3.2 for an example of coding CLS into CLIPS.

Unfortunately there are some disvantages that makes also this way unfeasible :

- A production system is apt to medium case of some inter related facts. This do not take advantage of the assumptions on the domains of CLS; instead, using an ad–hoc algorithm can exploit the tree structure of CLS term and the characteristic of abstract syntax CLS trees (like we have seen in 3.3.2), improving the performance.

- Production systems, created mainly in within of artificial intelligence, do not offer enough control on the conflict resolution strategy. In fact all analyzed expert systems offers a fixed set of resolution strategy (like depth, breath etc. . .), whereas we need to take control of all the conflict set and select which rule to apply regarding Gillespie's random procedure. This is the main impediment in the use of an already built production system inference engine.

3.3.3 Gillespie's Algorithm Extension to Deal with Rule Schemata

The Gillespie's direct method algorithm, as defined in [44] does not deals with variables and rate functions: each reaction is expressed by a ground rule with kinetic constant. We need instead to select a reaction among match set rule schemata; thus we have to deal with non ground rules with rate functions.

As we have seen in Section 2.2.1, in the papers of Barbuti et al. and in the Ph.D. thesis of Milazzo, at each step, the rule to be applied is chosen randomly with a probability that depends on an *actual application rate*. Such actual rate is the value obtained by the rate function multiplied by the number of possible positions (= context) in the term where the rule can be applied. The actual application rate is used also as the parameter of an exponential distribution to determine the quantity of time spent by the occurrence of the described event.

More precisely, at each step a set of *applicable ground rewrite rules* $AR(\mathcal{R}, T)$ is computed which contains all the ground rules that can be applied to T and that are obtained by instantiating variables in the rules in $\mathcal{R}$. In each of these ground rules we have $r = f(\sigma)$, where f is the rate function of the rewrite rule from which it was instantiated by means of an instantiation function σ. By the finiteness of $\mathcal{R}$ and of T we have that $AR(\mathcal{R}, T)$ is a finite set of ground rewrite rules. For each ground rule R in $AR(\mathcal{R}, T)$ and for each possible term T' that can be obtained by the application R, the number of different application positions in T where R can be applied producing T' is computed. Such a number, called the *application cardinality* of R leading from T to T', is denoted as $AC(R, T, T')$, and is the number that must be multiplied by the rate constant of R to obtain the actual application rate.

In other words, from the set of rules with variables are computed all the possible ground rules with the relative constant kinetic rate (obtained from the instantiation of variables that leads to this ground rule). Given that set we can use the standard Gillespie's algorithm as we do not have variables anymore.

Listing 3.4: Example of coding of CLS in CLIPS.
We can see the encoding in CLIP of term $(a)^L \rfloor (b \,|\, c \,|\, (b)^L \rfloor (c))$ and of rewrite rule $(\widetilde{x})^L \rfloor (X) \to \widetilde{x} \,|\, X$

```
///DEFINITIONS OF FACT TEMPLATE
deftemplate ELEMENT
        ( slot id (type symbols)          )
deftemplate COMPARTMENT
        ( multislot children (type factref) )
deftemplate SEQUENCE
        ( multislot childrens (type factref) )
deftemplate LOOP
        (
                slot membrane (type factref)
                slot content (type factref)
        )
///DEFINITION OF STARTING WORKING MEMORY
/// (= encoding of a term in clips's facts)
0 : (assert (ELEMENT (id "b")))
1 : (assert (ELEMENT (id "c")))
2 : (assert (LOOP (membrane 0 1 )))
3 : (assert (ELEMENT (id "b")))
4 : (assert (ELEMENT (id "c")))
5 : (assert (COMPARTMENT (children 2 3 4)))
6 : (assert (ELEMENT (id "a")))
7 : (assert (LOOP (membrane 6 content 5 )))
8 : (assert (COMPARTMENT (children 6 5)))
///DEFINITION OF RULES
(defrule R1
(?ref <- LOOP (membrane ?x content ?X) )
=>
(retract ?ref)
(assert (COMPARTMENT ?x ?X))
)
```

This approach is in practice not feasible because such set of ground rules is too large and must be recomputed at each step. In the implementation is therefore necessary to find directly the patterns with variables inside the term; where we have a match we have found a contexts of application of the rule schemata. Given the list of contexts in which each rule schemata is applicable, we have to choose directly from this list which rule and in which context has to be applied.

We now examine how to count occurrences of a rule with variable inside a term and then we present an extension of Gillespie's algorithm that deals with variables and rate functions. See Listings numbered 3.5 and 3.6 for the various version of Gillespie's algorithm.

Counting Occurrences of Rule Schemata Whereas in case of ground rules the count of occurrences of reactants in a solution is made as described in Section 1.4, making use of the product of binomial coefficients, dealing with variables we must fix how the variables influences this calculation. The naive approach is to multiply the product of the binomial coefficient by the number of the possible bindings of variables. Unfortunately this solution is a wrong interpretation of rule schemata. In fact, as shown in the following example, a rule can match in a node with some different bindings but the count of occurrences of that rule in that node is one. This is caused by the fact that all the different bindings, that allow a rule to match in a node, generate the same ground term as left hand side of the rule. Thus is necessary to group in a list these different bindings, concurring to a single match.

Example 3.5. Given the rule

$$a \cdot \widetilde{x} \,|\, a \cdot \widetilde{y} \overset{k}{\mapsto} b$$

in which the left hand side pattern represents a set of reactants. In the term

$$a \cdot b \,|\, a \cdot c$$

there is only one combination of the reactants that can be obtained with different instantiation of variables. In fact the all the possible binding are

$$\sigma_1 = \{\widetilde{x} = b\,,\, \widetilde{y} = c\} \;\; \sigma_2 = \{\widetilde{x} = c\,,\, \widetilde{y} = b\}$$

both of them give the following ground rule

$$a \cdot b \,|\, a \cdot c \overset{k}{\mapsto} b$$

But we must be aware of the fact that, even if different bindings of a match of a rule generate through instantiation the same left hand side, this is not true for the right hand side of the rule (and also for the value obtained from the rate function as we see next).

Example 3.6. Given the rule

$$a \cdot \widetilde{x} \,|\, a \cdot \widetilde{y} \overset{k}{\mapsto} b \,|\, \widetilde{x}$$

an the same term of the previous example even if the number of times in which the reactants are present in the solution is one, the instantiation of the right hand side of the rule gives two different right hand side: $b \,|\, b$ and $b \,|\, c$.

Thus we must introduce the stochastic component also in the selection of the binding. That is: we store each match with all the possible bindings counting it as a single occurrence of the reactants; then, when we have chosen an occurrence, the binding used to obtain rate constant from rate function and to obtain the right hand side of the rule are chosen randomly among the possible different bindings. These bindings gives the same occurrence of the reactants thus are considered equivalents and selected randomly.

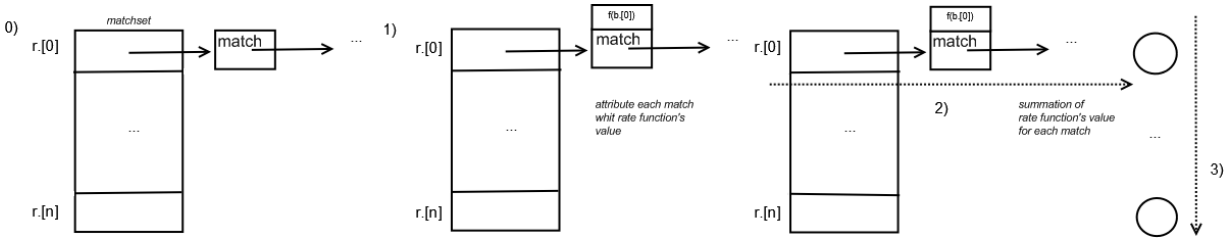


Figure 3.11: Schematic of how to extend the matchset of rule schemata in order to apply correctly the Gillespie's SSA.

Application of Gillespie's SSA to Matchset of Rule Schemata In order to do this we execute the following steps (see Figure 3.11).

1. Given an array containing a list of matches for each rule, we enrich each match with the value given by the rate function of the rule applied with one of any of the possible bindings for that match. Practically this is obtained using the F# library function `map`[9] whit a function that transforms each list of matches in a list of pairs (match, value of rate function).

2. In order to obtain a rate value for each rule schema, we do the summation of all the rate values associated with the matches. Practically this is obtained using the F# library function `fold_left`[10] on each list of matches.

3. Now, having all rates as constants, like the Gillespie's SSA selects a possible reaction, we select randomly and execute a match of a rule schema. We get a random number from 0 to the summation of the rate of each schema. This number is used to select a rule schema with a probability proportional to its rate. Similarly this number is also used to select a match among those of the chosen schema, with a probability proportional to the rates computed in 1). Given the selected match, we choose randomly what binding to use, among the set of its possible bindings.

As we have discussed in previous paragraph, different bindings originate the same ground left hand side; thus they must be considered as a single occurrence of the rule schema. But there is an inconvenient: even if different possible bindings give

[9] `map : ('a -> 'b) -> 'a list -> 'b list`
[10] `fold_left : ('b -> 'a -> 'b) -> 'b -> 'a list -> 'b`

the same ground left hand side, it is possible that they give different ground right hand side and different value of rate function.

Example 3.7. If we have

$$f(b) = if\, occ(c, X) = 1\, then\, 1\, else\, 0$$

and the rule

$$a \,|\, X \overset{f(b)+1}{\mapsto} a \,|\, a \,|\, X$$

applied to term

$$a \,|\, b \,|\, c$$

we can bind X to b or c or $b \,|\, c$ or ε, for the same instance of the rule, originating different value of rate function. In fact in the values can be 0 or 1.

This represents a defect of the CLS formalism; in fact the value of arbitrary rate functions computed on possible different bindings can give different value. Thus in the implementation of the simulator we assume that the rate functions are defined in such way that they give the same value for all the bindings that give the same ground term as instantiation of the left hand side of the correspondent rule. With this assumption on the rate functions we can always select the first binding.

This ambiguity can be avoided using a variant of CLS called CLS+ (see Section 6.2.2).

3.3.4 Compilation and Execution of C# Code in the Rate Functions

As regards the definition of rate functions of the SCLS rules (see Section 2.4) we have chosen to allow the user to write rate functions in the simulation input file by using the C# [53][11] programming language.

In this way we have the following advantages :

- such a programming language is Turing complete, and the users can use all the .NET library function. Moreover it is possible to develop a library of additional helper functions and then use it: we should simply register the library in the .NET *Global Assembly Cache* and then call it in the rate function source. As example we have developed a function that counts the occurrence of certain elements inside certain variable's instantiations; this function, called `occ`, has the same semantics of that seen in the Example 2.10 (see Section 4.4 for an use example).

- we can use the dynamic features of .NET rendering the implementation very simple. In fact the *CodeDom* and the *Reflection*, features of .NET infrastructure, give you the ability to dynamically build C# code into a string, compile it in memory, and run it, by program.

[11] We would have intentional to use F#, that have a lighter syntax, but the corresponding *CodeDom* it is not still mature.

That is (see Figure 3.12 and Listing 3.7 for details).

We take the rate function source and put it into a the code of a class definition: this class has a method that take the given binding and execute the (pre–processed[12]) source of the user specified rate function.

A different class is compiled for each rule and an its instance is placed into the rule.

At the time in which we need to evaluate the rate function we need only to call the method using as parameter the selected instantiation (= a binding) of the rule.

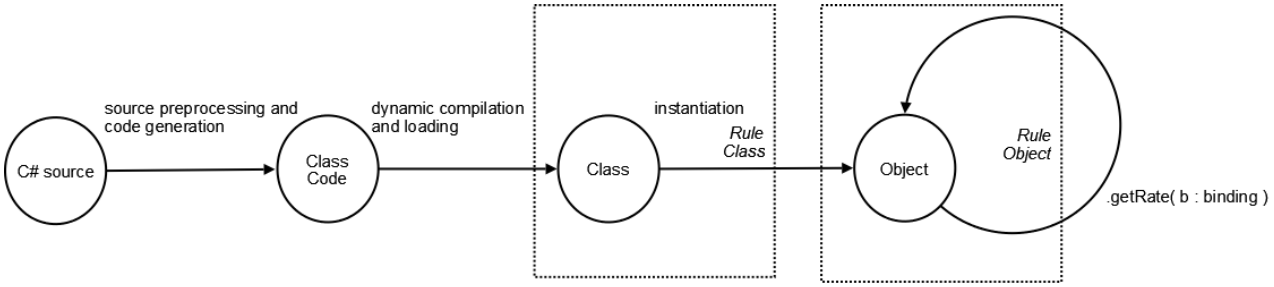


Figure 3.12: Rate functions dynamic code generation, compilation and execution.

[12]For example we replace the call to `abs` in `Math.abs` and give the access to the scope of some helper function like `occ` seen in Example 2.10 .

Listing 3.5: Gillespie's stochastic simulation algorithm.

```
/// Main simulation's procedure
let Simulate(start_clock: float, stop_clock: float, model: Model)
   =
    let mutable clock = start_clock
    while (clock < stop_clock) do
        /// get the set of applicable reactions
        let matchset = get_matches model.term model.rules
        /// select one of these according to Gillespie's SSA
        let (tau, rule , m) = Gillespie(matchset, model.rules)
        if ( tau > 0 )
                /// there is selected some reaction;
                /// apply the reaction and spend the employed time
                then model.term.replace(
                        m,
                        ((model.rules).(ruleNumber)).left,
                        ((model.rules).(ruleNumber)).right
                )
                /// there are not any possible reactions and
                /// there will not even more (simulation can be
                   stopped)
                else break
        clock <- !clock + tau
    done

/// Gillespie's SSA
/// rules is the array of possible reactions
/// matchset is the array of list of matches for each rule
and Gillespie (matchset: Match array, rules: rules array) =
    let par = count_par matchset rules
    let (tau:float) =
                if (par = 0)
                        then 0
                        else - log(rnd.NextDouble()) / par
    let rule,m = getRuleAndMatch(rnd.NextDouble(),matchset)
    (tau, rule, m)

/// compute the summation from i=0 to rules.Lenght
///     of ( matchset.[i].Lenght * rules.[i].kinetic)
and count_par matchset rules =
    let i = ref -1
    let f = fun acc rule -> i:= !i + 1; acc + (rule.kinetic *
       matchset.[i].Lenght)
    Array.fold_left f rules
```

Listing 3.6: Gillespie's algorithm with variables and rate functions.

```
let Simulate(start_clock: float, stop_clock: float, model: Model) =
    let mutable clock = start_clock
    let mutable simulate = true
    while   (clock < stop_clock) && simulate do
        let matchset = get_matches model.term model.rules
        let (tau, ruleNumber, matchNumber, augmented_matchset) = Gillespie(matchSet, model
            .rules)
        if( tau > 0 )
                then
                    let selected_match = getNthMatchFromAumentedMatchList ( matchNumber,(
                        augmented_matchset.[ruleNumber]))
                    let left_hand_side,right_hand_side  =
                        if  (((model.rules).(ruleNumber))).IsGround
                            then    (((model.rules).(ruleNumber)).left),(((model.rules).(
                                ruleNumber)).right)
                            else
                                    let bindings_number =  rnd.Next(selected_match.
                                        bind_list.Length - 1)
                                    let selected_bind = List.nth selected_match.bind_list
                                        bindings_number
                                    Node.instantiateNode( (((model.rules).(ruleNumber)).
                                        left :> Node) , selected_bind,
                                    Node.instantiateNode((((model.rules).(ruleNumber)).
                                        right :> Node) , selected_bind)
                                    model.term.Replace(
                                      selected_match.where,
                                      left_hand_side,
                                      right_hand_side
                                    )
                else    simulate <- false
    clock := !clock + tau

and Gillespie ((matchset: ResizeArray<Match> array), (rules: SCLS.Rule array)) =
    let augmented_matchset,par = countPar_ratefunction(matchset, rules)
    let (tau:float) =  if (par = double 0) then double 0 else - log(rnd.NextDouble()) /
        par
    let (r:float) = rnd.NextDouble()
    let ruleNumber,matchNumber = getRuleNumberFromMatchNumber(r,augmented_matchset)
    (tau, ruleNumber, matchNumber, augmented_matchset)

and getRuleNumberFromMatchNumber ((absoluteMatchNumber: float) , (augmented_matchset:(
    ResizeArray<(Match * float)> * float) array ) ) :int * float =
    let acc = ref 0.0
    let index = Array.find_index (
                        fun (a,b) ->
                                (acc := !acc + b);
                                if absoluteMatchNumber <= !acc then acc:=!acc-b; true else
                                    false
                       ) augmented_matchset
    (index, absoluteMatchNumber - !acc)

and countPar_ratefunction ((matchset: ResizeArray<Match> array), (rules: SCLS.Rule array))
    =
    let m =
        attributat_matchset matchset
        |> Array.map (fun l -> (l,compute_acc_for_attributated_matchlist l))
    let par = Array.fold_left (fun acc (m,f) -> acc + f) 0.0 m
    m,par

and attribute_match =
    fun (m:Match) ->
        /// associate at each match the rate function of the rule
        /// computed on the first bindings
        m,(rules.[m.rule_id]).getRate((List.hd m.bind_list)) * float m.repetitions

and attribute_matchlist =
    fun (ml: ResizeArray<Match>) ->
    ResizeArray.map attribute_match ml

and attributate_matchset matchset =
    Array.map attribute_matchlist matchset

and compute_acc_for_attributated_matchlist =
    fun (aml:ResizeArray<(Match * float)>) ->
    ResizeArray.fold_left (fun (acc:float) (m:Match,f:float) -> acc + f ) 0.0 aml
```

Listing 3.7: Dynamic rate function evaluator

```
  public class CSharpCodeExpressionEvaluator
{
    public object myobj = null;
    public ArrayList errorMessages;

    public CSharpCodeExpressionEvaluator()
    {
        errorMessages = new ArrayList();
    }
    public bool init(string expr)
    {
        Environment.CurrentDirectory =
                            System.AppDomain.CurrentDomain.BaseDirectory;
        Microsoft.CSharp.CSharpCodeProvider cp =
                            new Microsoft.CSharp.CSharpCodeProvider();
        System.CodeDom.Compiler.CompilerParameters cpar =
                            new System.CodeDom.Compiler.CompilerParameters();
        cpar.GenerateInMemory = true;
        cpar.GenerateExecutable = false;
        cpar.ReferencedAssemblies.Add("system.dll");
        cpar.ReferencedAssemblies.Add("scls.dll");
        string src =
        @"using System;
          using SCLSm;
          using SCLSm.Occurrences;
          using SCLSm.SCLS;
          class myclass
          {
            public myclass(){}
            public static double eval(bindings b)
            {
                return (" + expr + "); " +
           @"}
         } ";
        System.CodeDom.Compiler.CompilerResults cr =
                            cp.CompileAssemblyFromSource(cpar, src);
        foreach (System.CodeDom.Compiler.CompilerError ce in cr.Errors)
                    { errorMessages.Add(ce.ErrorText); }
        if (cr.Errors.Count == 0 && cr.CompiledAssembly != null)
        {
            Type ObjType = cr.CompiledAssembly.GetType("myclass");
            try
            {
                if (ObjType != null)
                {
                    myobj = Activator.CreateInstance(ObjType);
                }
            }
            catch (Exception ex)
            {
                errorMessages.Add(ex.Message);
            }
            return true;
        }
        else
            return false;
    }
    public double evalRateFunction(SCLSm.SCLS.bindings binding)
    {
        double val = 0.0;
        Object[] myParams = new Object[1] { binding};
        if (myobj != null)
        {
            System.Reflection.MethodInfo evalMethod =
                                    myobj.GetType().GetMethod("eval");
            val = (double)evalMethod.Invoke(myobj, myParams);
        }
        return val;
    }
}
```

Part III

RESULTS

Chapter 4

Use Cases

In this chapter we present some simulation results obtained with the developed simulator.

To test the correctness of the engine we have ran some well known examples without variables, just simulated with other simulator, like Lotka-Volterra (4.1) and Brussellator (4.2). Moreover, to compare the results of simulation with experimental data obtained by biologists, we have simulated the reactions related to the activity of the Sorbytol Dehydrogenase enzyme in the calf eye (4.3).

To test the correctness of stochastic algorithm with variables, rate functions and rule schemata we have run more complex examples like the genetic regulation process of lactose operon in *Escherichia coli* (4.4) and the quorum sensing phenomenon in *Pseudomas aeruginosa* (4.5).

4.1 Lotka–Volterra

The Lotka–Volterra equations, also known as the predator-prey equations, are a pair of first order, non-linear, differential equations frequently used to describe the dynamics of biological systems in which two species interact, one a predator and one its prey. They were proposed independently by Alfred J. Lotka in 1925 and Vito Volterra in 1926.

The following Stochastic CLS rules model the dynamic of the system:

$$Y_1 \overset{k_1}{\mapsto} Y_1 \,|\, Y_1$$

$$Y_1 \,|\, Y_2 \overset{k_2}{\mapsto} Y_2 \,|\, Y_2$$

$$Y_2 \overset{k_3}{\mapsto} \varepsilon$$

where $k_1 = 10$, $k_2 = 0.1$ and $k_3 = 10$. In Figure 4.1 we show two simulation run where in both the initial term contains the parallel composition of 100 Y_1 and 100 Y_2.

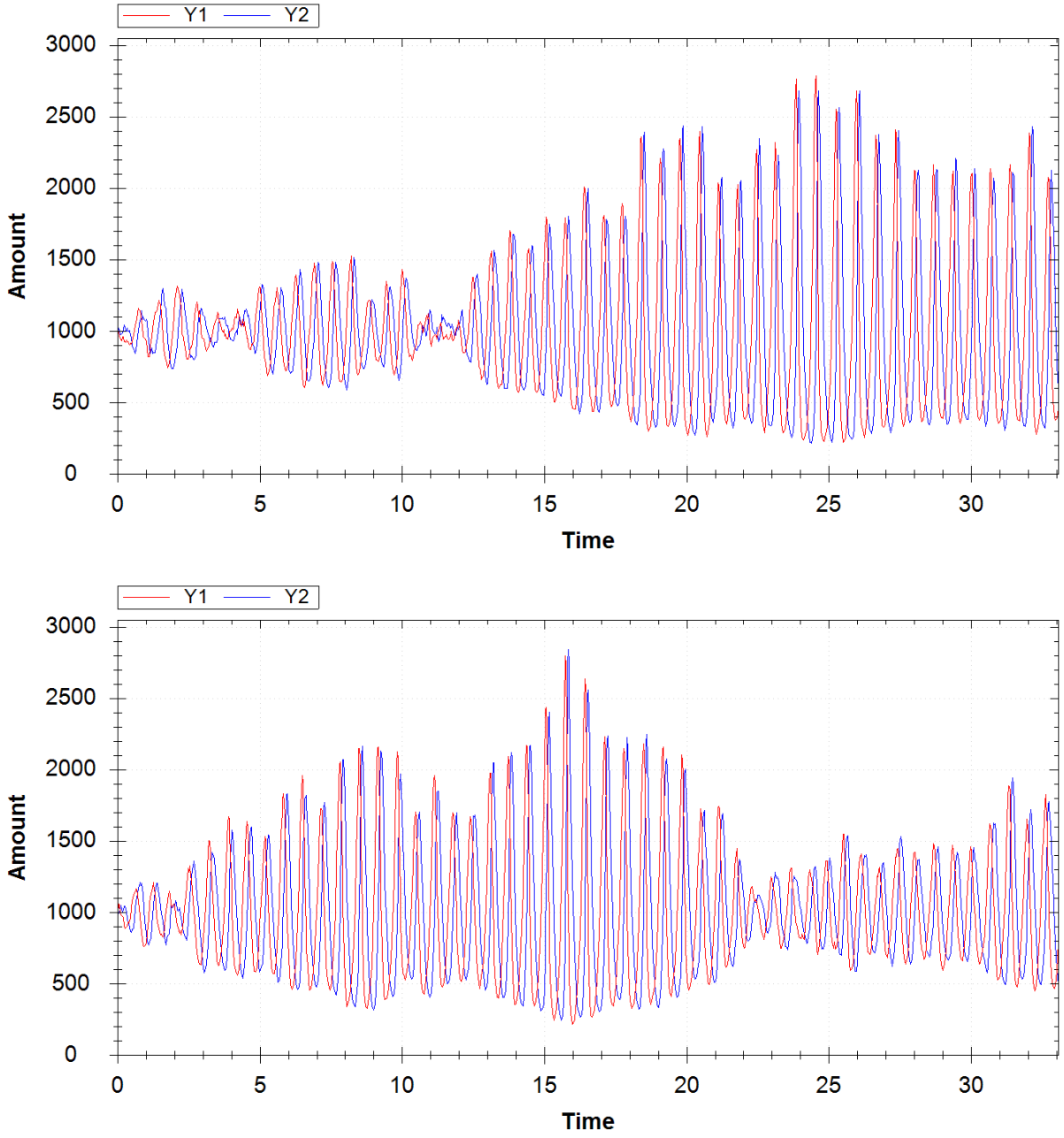


Figure 4.1: Two runs of Lotka Volterra simulation. The time is expressed in seconds. We can see as the simulations evolve like attended and how the stochasticity of simulation algorithm makes two successive run to follow different trajectory.

4.2 Brussellator

The Brussellator is a model of enzymatic chemical reaction.

The following Stochastic CLS rules model the dynamic system :

$$X \overset{k_1}{\mapsto} X \,|\, Y$$

$$Y_1 \overset{k_2}{\mapsto} Y_2$$

$$Y_1 \,|\, Y_1 \,|\, Y_2 \overset{k_3}{\mapsto} Y_1 \,|\, Y_1 \,|\, Y_1$$

$$Y_1 \overset{k_4}{\mapsto} \varepsilon$$

where $k_1 = 5000$, $k_2 = 50$,$k_3 = 0.00005$ and $k_4 = 5$. In Figure 4.2 we show the result of a simulation where the initial term is[1] $X \,|\, Y_1 \times 1000 \,|\, Y_2 \times 2000$

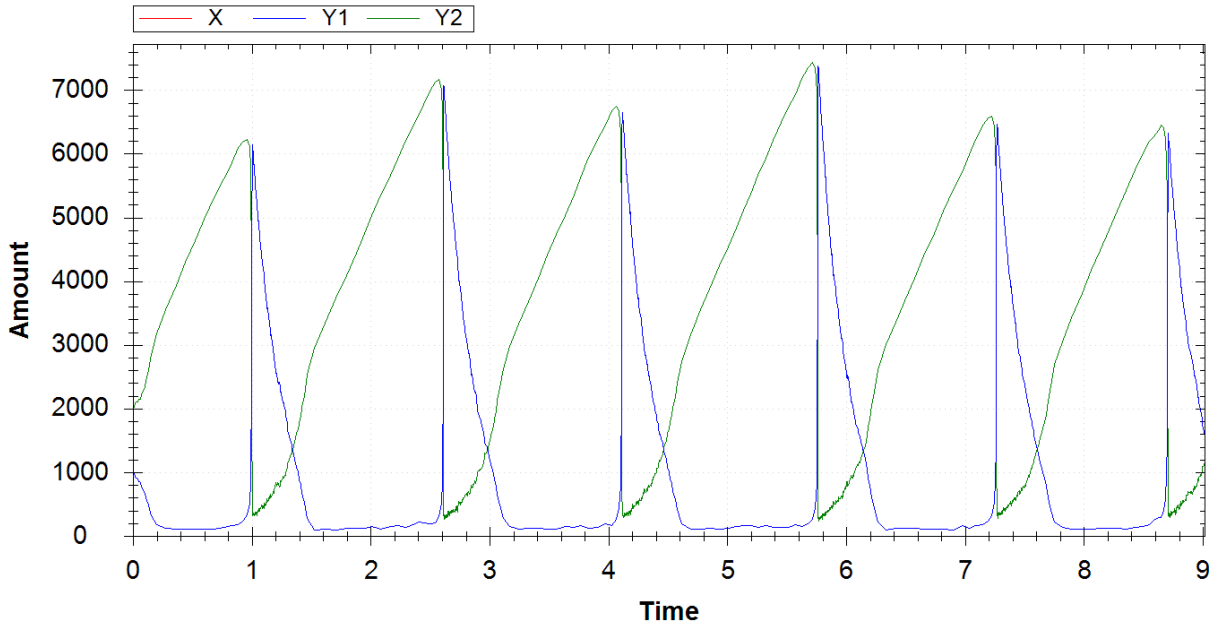


Figure 4.2: Simulation result of Brussellator

[1] Where $n \times T$ stands for a parallel composition $T \,|\, \ldots \,|\, T$ of length n

4.3 Sorbitol Dehydrogenase (SDH)

A more complex example is that related to the activity of the sorbytol dehydrogenase enzyme in the calf eye [72] that has already studied, with different techniques, in [16, 18]. The enzyme sorbytol dehydrogenase (SDH) catalyzes the reversible oxidation of Sorbitol and other polyalcohols to the corresponding keto–sugars (the accumulation of sorbytol in the calf eye has been proposed as the primary event in the development of sugar cataract in the calf). The rewrite rules modeling the reactions are shown in the following scheme:

$$E \,|\, NADH \underset{k_2}{\overset{k_1}{\rightleftharpoons}} ENADH \qquad ENADH \,|\, F \underset{k_4}{\overset{k_3}{\rightleftharpoons}} ENAD^+ \,|\, S$$

$$ENAD^+ \underset{k_6}{\overset{k_5}{\rightleftharpoons}} E \,|\, NAD^+ \qquad E \overset{k_7}{\rightarrow} \varepsilon$$

where E represents the enzyme sorbitol dehydrogenase, S and F represent sorbitol and fructose, respectively, $NADH$ represents the nicotinamide adenine dinucleotide and NAD^+ is the oxidized form of $NADH$; $k_1, \ldots, k_7$ are the kinetic constants. Note that the enzyme degradation is modeled by the transformation of E into ε.

In Figure 4.3 is show the result of simulating that system with $k_1 = 0.0000062$, $k_2 = 33$,$k_3 = 0.00005$, $k_3 = 0.000000002$, $k_4 = 227$, $k_5 = 50$,$k_6 = 0.0000006$ and $k_7 = 0.0019$ starting with term

$$E \times 210 \,|\, F \times (4 * 10^{11}) \,|\, NADH \times (16 * 10^7)$$

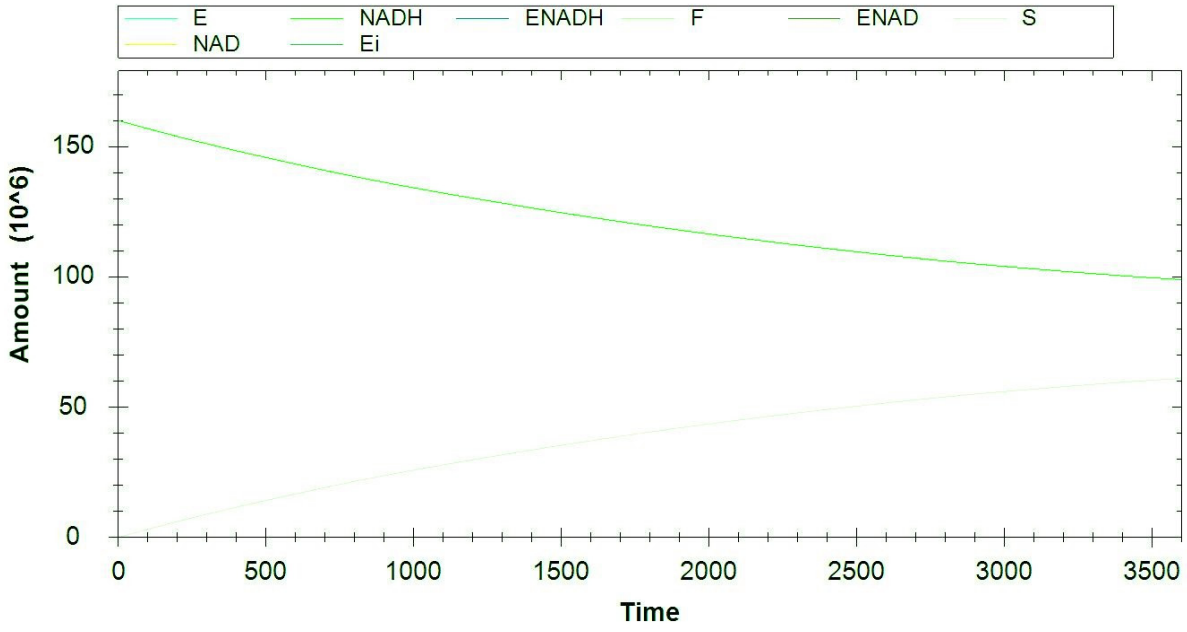


Figure 4.3: Simulation result of Sorbitol Dehydrogenase.

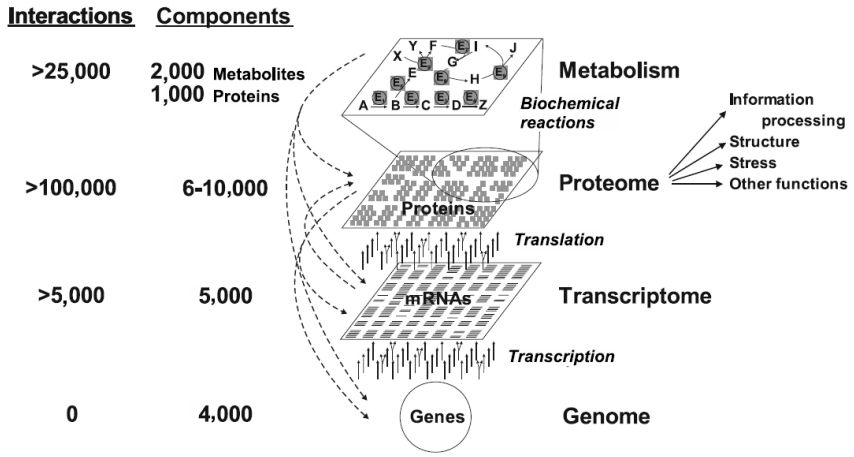


Figure 4.4: Example of complexity in networks of *Escerichia coli*. Regulatory interactions are indicated by dashed lines. Transcript interactions are based on operon structures and ribosomal RNA interactions. Proteome interactions include an average of 6-7 protein–protein interactions as well as protein–DNA, protein–RNA, and protein–membrane interactions. Metabolic interactions include biochemical transformations and regulatory among metabolites, RNA, and protein. Protein number encompass differences in folding, size, and covalent modifications (note that not all proteins are necessarily present at the same time).

The results are the same of with those described in [17], that in its turn are the in agreement with experimental results.

4.4 Lactose Operon in *Escherichia coli*

The *Escherichia Coli* is one of the simplest bacteria and thus one of the most studied. Although its relative simplicity, if compared with other organisms, the simulation of the interactions among its component is not oblivious and involves a great number of interrelated circuits (see Figure 4.4).

In this section a Stochastic CLS model of the regulation process of the lactose operon in *Escherichia coli* (E. coli) is presented. We use our simulator of the Stochastic CLS to analyze the gene regulation process in different situations.

Gene Regulation Process As most bacteria, E.coli reacts to changes in its environment through changes in the kinds of enzymes it produces. In order to save energy, bacteria do not synthesize degradative enzymes unless the substrates for these enzymes are present in the environment. For example, E. coli does not synthesize the enzymes that degrade lactose unless lactose is in the environment. This phenomenon is called *enzyme induction* or, more generally, *gene regulation* since it is obtained by controlling the transcription of some genes into the corresponding enzymes.

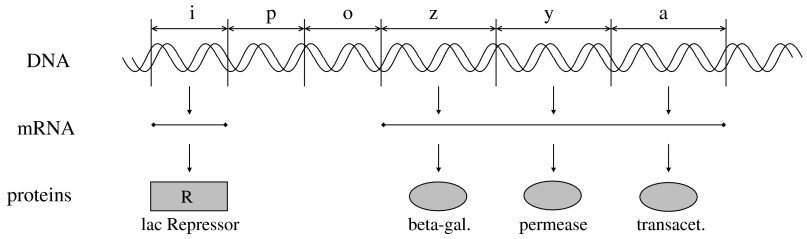


Figure 4.5: The lactose operon.

We consider the lactose degradation example in E. coli. Two enzymes are required to start the breaking process: the *lactose permease*, which is incorporated in the membrane of the bacterium and actively transports the sugar into the cell, and the *beta galactosidase*, which splits lactose into glucose and galactose. The bacterium produces also the *transacetylase* enzyme, whose function is less known, but is surely related with the usage of lactose.

The sequence of genes in the DNA of E. coli which produces the described enzymes, is known as the *lactose operon* (see Figure 4.5). It is composed by six genes: the first three (i, p, o) regulate the production of the enzymes, and the last three (z, y, a), called *structural genes*, are transcribed (when allowed) into the mRNA for beta galactosidase, lactose permease and transacetylase, respectively.

The regulation process is as follows (see Figure 4.6): gene i encodes the *lac Repressor*, which in the absence of lactose, binds to gene o (the *operator*). Transcription of structural genes into mRNA is performed by the RNA polymerase enzyme, which usually binds to gene p (the *promoter*) and scans the operon from left to right by transcribing the three structural genes z, y and a into a single mRNA fragment. When the lac Repressor is bound to gene o, it becomes an obstacle for the RNA polymerase, and transcription of the structural genes is not performed. On the other hand, when lactose is present inside the bacterium, it binds to the Repressor and this cannot stop any more the activity of the RNA polymerase. In this case the transcription is performed and the three enzymes for lactose degradation are synthesized.

Stochastic CLS Model A detailed mathematical model of the regulation process can be found in [105]. It includes information on the influence of lactose degradation on the growth of the bacterium.

In work by Milazzo is shown a Stochastic CLS model of the gene regulation process, with stochastic rates taken from [104]. The membrane of the bacterium is modeled as the looping sequence $(m)^L$, where the elementary constituent m generically denotes the whole membrane surface in normal conditions. Moreover, the lactose operon is modeled as the sequence $lacI \cdot lacP \cdot lacO \cdot lacZ \cdot lacY \cdot lacA$ ($lacI{-}A$ for short), in which each element corresponds to a gene. We replace $lacO$ with RO in the sequence when the lac Repressor is bound to gene o, and $lacP$ with PP when the RNA polymerase is bound to gene p. When the lac Repressor and

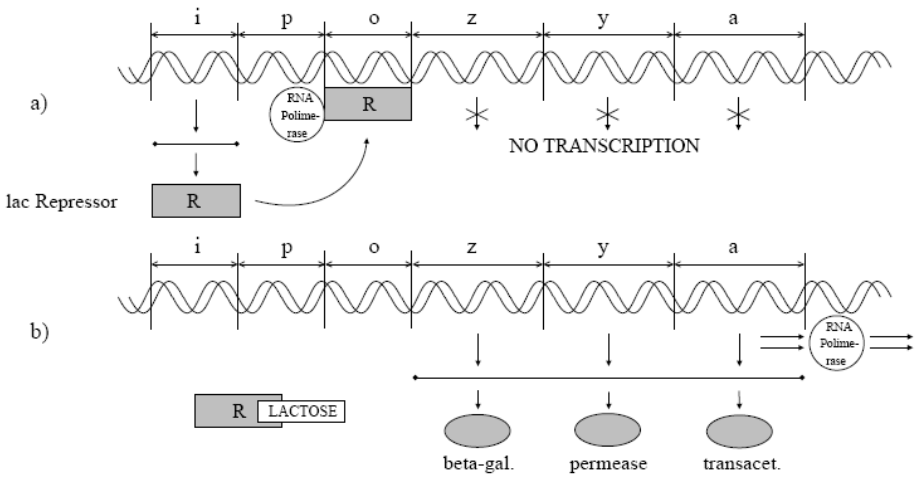


Figure 4.6: The regulation process. In the absence of lactose (case a) the lac Repressor binds to gene o and precludes the RNA polymerase from transcribing genes z, y and a. When lactose is present (case b) it binds to and inactivates the lac Repressor.

the RNA polymerase are unbound, they are modeled by the elementary constituents *repr* and *polym*, respectively. The mRNA of the lac Repressor is modeled as the elementary constituent $Irna$, a molecule of lactose as the elementary constituent $LACT$, and beta galactosidase, lactose permease and transacetylase enzymes as elementary constituents *betagal*, *perm* and *transac*, respectively. Finally, since the three structural genes are transcribed into a single mRNA fragment (see Fig. 4.5), such mRNA is modeled as a single elementary constituent *Rna*.

The initial state of the bacterium when no lactose is present in the environment is modeled by the following term (where $n \times T$ stands for a parallel composition $T \,|\, \ldots \,|\, T$ of length n):

$$Ecoli \;::=\; \big(m\big)^{L} \rfloor \, (lacI\text{–}A \,|\, 30 \times polym \,|\, 100 \times repr) \tag{4.1}$$

The presence of lactose in the environment is modeled by composing $Ecoli$ in parallel with a number of $LACT$ elements as follows:

$$EcoliLact \;::=\; Ecoli \,|\, 10000 \times LACT \tag{4.2}$$

The transcription of the DNA, the binding of the lac Repressor to gene o, and the interaction between lactose and the lac Repressor are modeled by the following set of rule schemata:

$$lacI \cdot \widetilde{x} \;\overset{0.02}{\longmapsto}\; lacI \cdot \widetilde{x} \,|\, Irna \tag{S1}$$

$$Irna \;\overset{0.1}{\longmapsto}\; Irna \,|\, repr \tag{S2}$$

$$polym \,|\, \widetilde{x} \cdot lacP \cdot \widetilde{y} \;\overset{0.1}{\longmapsto}\; \widetilde{x} \cdot PP \cdot \widetilde{y} \tag{S3}$$

$$\widetilde{x} \cdot PP \cdot \widetilde{y} \;\overset{0.01}{\longmapsto}\; polym \,|\, \widetilde{x} \cdot lacP \cdot \widetilde{y} \tag{S4}$$

$$\widetilde{x} \cdot PP \cdot lacO \cdot \widetilde{y} \;\overset{20.0}{\longmapsto}\; polym \,|\, Rna \,|\, \widetilde{x} \cdot lacP \cdot lacO \cdot \widetilde{y} \tag{S5}$$

$$Rna \overset{0.1}{\longmapsto} Rna \,|\, betagal \,|\, perm \,|\, transac \tag{S6}$$

$$repr \,|\, \widetilde{x} \cdot lacO \cdot \widetilde{y} \overset{1.0}{\longmapsto} \widetilde{x} \cdot RO \cdot \widetilde{y} \tag{S7}$$

$$\widetilde{x} \cdot RO \cdot \widetilde{y} \overset{0.01}{\longmapsto} repr \,|\, \widetilde{x} \cdot lacO \cdot \widetilde{y} \tag{S8}$$

$$repr \,|\, LACT \overset{0.005}{\longmapsto} RLACT \tag{S9}$$

$$RLACT \overset{0.1}{\longmapsto} repr \,|\, LACT \tag{S10}$$

Schemata (S1) and (S2) describe the transcription and translation of gene i into the lac Repressor (assumed for simplicity to be performed without the participation of the RNA polymerase). Schemata (S3) and (S4) describe the binding (and unbinding) of the RNA polymerase to gene p. Schemata (S5) and (S6) describe the transcription and translation of the three structural genes. Transcription of such genes can be performed only when the sequence contains $lacO$ instead of RO, that is when the lac Repressor is not bound to gene o. Schemata (S7) and (S8) describe the binding and unbinding, respectively, of the lac Repressor to gene o. Finally, schemata (S9) and (S10) describe the binding and unbinding, respectively, of the lactose to the lac Repressor.

The following schemata describe the behavior of the three enzymes for lactose degradation:

$$\left(\widetilde{x}\right)^L \rfloor (perm \,|\, X) \overset{0.1 \cdot f_1}{\longmapsto} \left(perm \cdot \widetilde{x}\right)^L \rfloor X \tag{S11}$$

$$LACT \,|\, \left(perm \cdot \widetilde{x}\right)^L \rfloor X \overset{0.0001 \cdot f_2}{\longmapsto} \left(perm \cdot \widetilde{x}\right)^L \rfloor (LACT|X) \tag{S12}$$

$$betagal \,|\, LACT \overset{0.00001}{\longmapsto} betagal \,|\, GLU \,|\, GAL \tag{S13}$$

where $f_1(\sigma) = occ(perm, \sigma(X)) + 1$, $f_2(\sigma) = occ(perm, \sigma(\widetilde{x})) + 1$ and $occ(a, T)$ is the same of Example 2.10 [2] .

Schema (S11) describes the incorporation of the lactose permease in the membrane of the bacterium, schema (S12) the transportation of lactose from the environment to the interior performed by the lactose permease, and schema (S13) the decomposition of the lactose into glucose (denoted GLU) and galactose (denoted GAL) performed by the beta galactosidase.

The following schemata describe degradation of all the proteins and pieces of mRNA involved in the process:

$$perm \overset{0.001}{\mapsto} \varepsilon \tag{S14}$$

$$betagal \overset{0.001}{\mapsto} \varepsilon \tag{S15}$$

$$transac \overset{0.001}{\mapsto} \varepsilon \tag{S16}$$

$$repr \overset{0.002}{\mapsto} \varepsilon \tag{S17}$$

$$Irna \overset{0.01}{\mapsto} \varepsilon \tag{S18}$$

$$Rna \overset{0.01}{\mapsto} \varepsilon \tag{S19}$$

$$RLACT \overset{0.002}{\mapsto} LACT \tag{S20}$$

We recall that sequences are not allowed as context of application of the rules, hence the rule derived from schema (S14) cannot be applied to $perm$ when it is an element of the looping sequence representing the membrane of the bacterium. This motivates the presence of the following final schema:

$$\left(perm \cdot \widetilde{x}\right)^L \rfloor X \overset{0.001 \cdot f_2}{\longmapsto} \left(\widetilde{x}\right)^L \rfloor X \tag{S21}$$

[2] In this use case we can see the use of rate functions that give different value according to the selected instantiation. The simulator accepts f_1 as `(occ(perm,X)+1)` and f_2 as `(occ(perm,x)+1)`.

Simulation Results We simulated the evolution of the bacterium in the absence of lactose (modeled by the term *Ecoli* of Eq. (4.1)) and in the presence of 100 molecules of lactose in the environment (modeled by the term *EcoliLact* of Eq. (4.2)). The evolution of the two terms is given by the application of the set of rewrite rule schemata $\{(S1), \ldots, (S21)\}$.

In Figure 4.7 we show the results of simulation when the lactose is absent. In this situation the lac Represson inhibits the transcription of the enzymes. In fact in the plot is shown that those enzymes (betagal. and perm.) do not exceed amount of some tens, whereas the lac Repressor oscillates around the thousands units.

In Figure 4.8 we show the results of the simulation when the lactose is present in the environment. In this simulations the production of the beta galactosidase and lactose permease enzymes start almost immediately.

We remark that the different times in the production of enzymes in the various simulations is not significant; in fact, the amount of time elapsed before the production of these enzymes does not depend on the presence of the lactose in the environment, because the lactose cannot enter the bacterium until some molecule of permease has been incorporated in the membrane.

Once some molecule of lactose permease joins the membrane, the lactose starts entering the bacterium. In fact, the graph on Figure 4.9 and 4.10 is shown that the number of molecules in the environment rapidly decreases. Once entered, the lactose interacts with the lac Repressor: the lac Repressors bind to lactose (indicated with RLAC). In this situation the production of the beta galactosidase and lactose permease enzymes is favored and they are present in hundreds. At this stage, the lactose is decomposed by the beta galactosidase and the production of glucose starts.

Once all the molecules of lactose have been decomposed, the number of lac Repressors gradually increases, reaching the same values of the first simulation (see Figure 4.7) and halting the process.

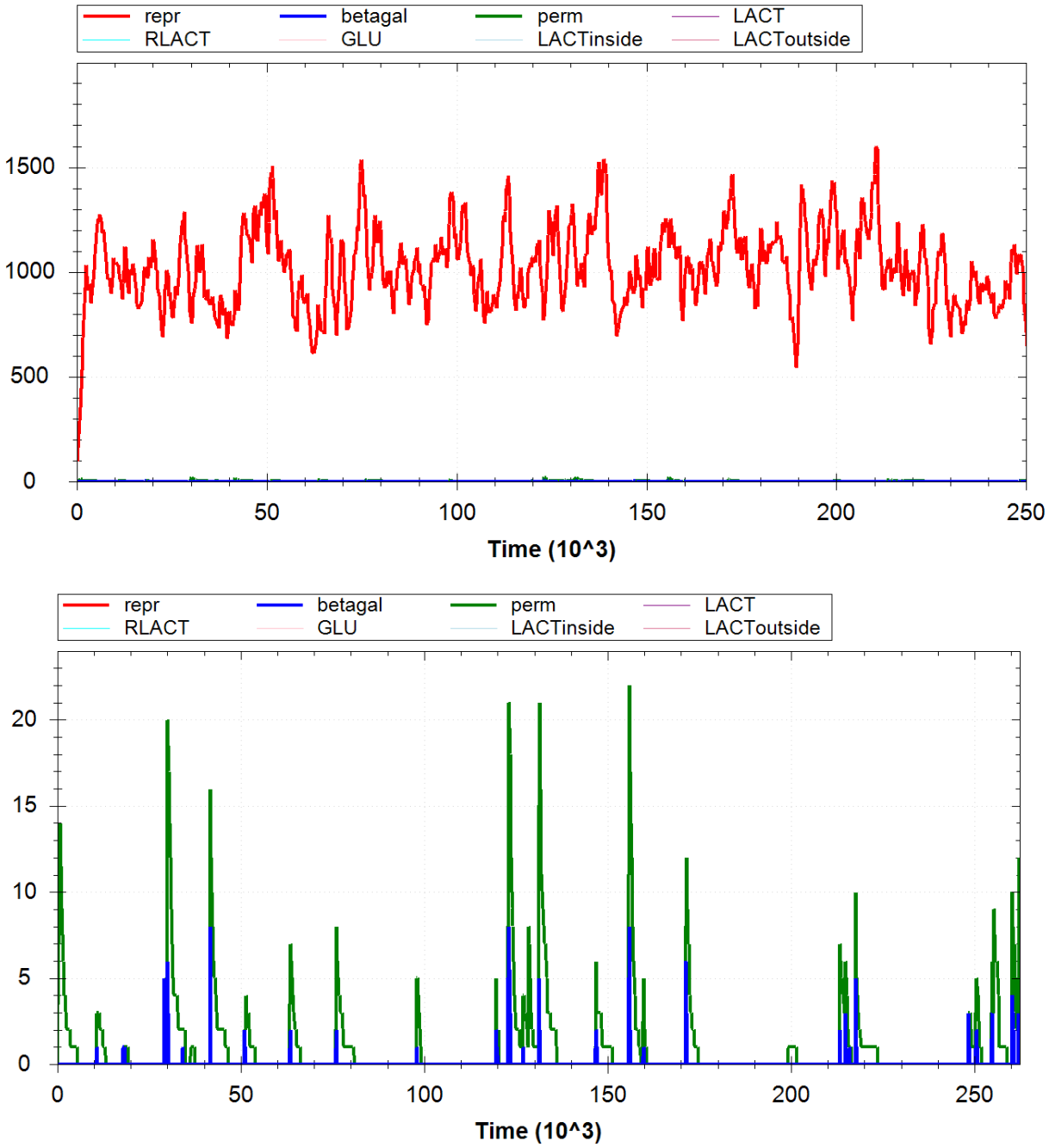


Figure 4.7: Result of simulation of the regulation process of lactose operon in *Escherichia coli* in absence of lactose.

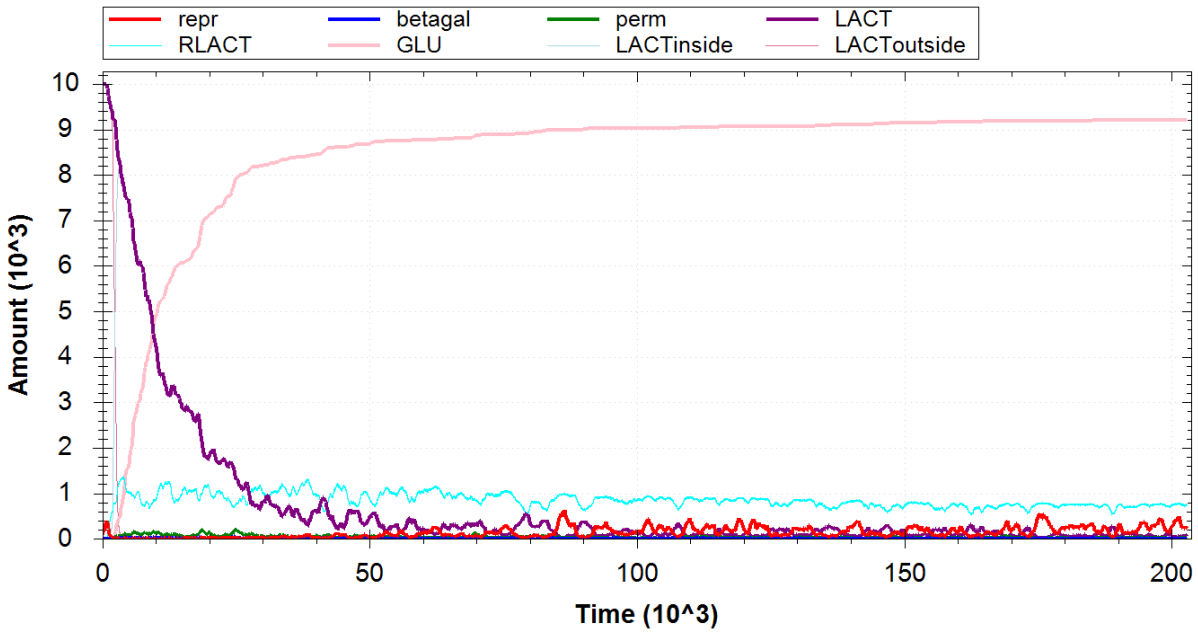


Figure 4.8: Results of simulation of the regulation process of lactose operon in *Escherichia coli* when lactose is present in the environment.

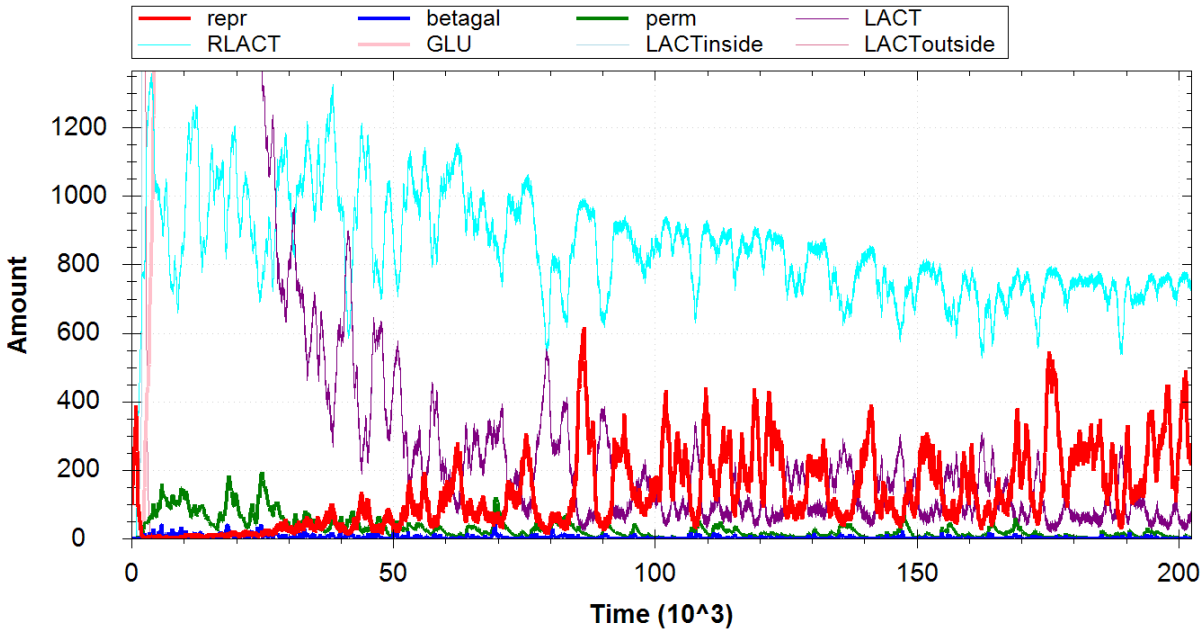


Figure 4.9: Zoom on the results of simulation of the regulation process of lactose operon in *Escherichia coli* when lactose is present in the environment.

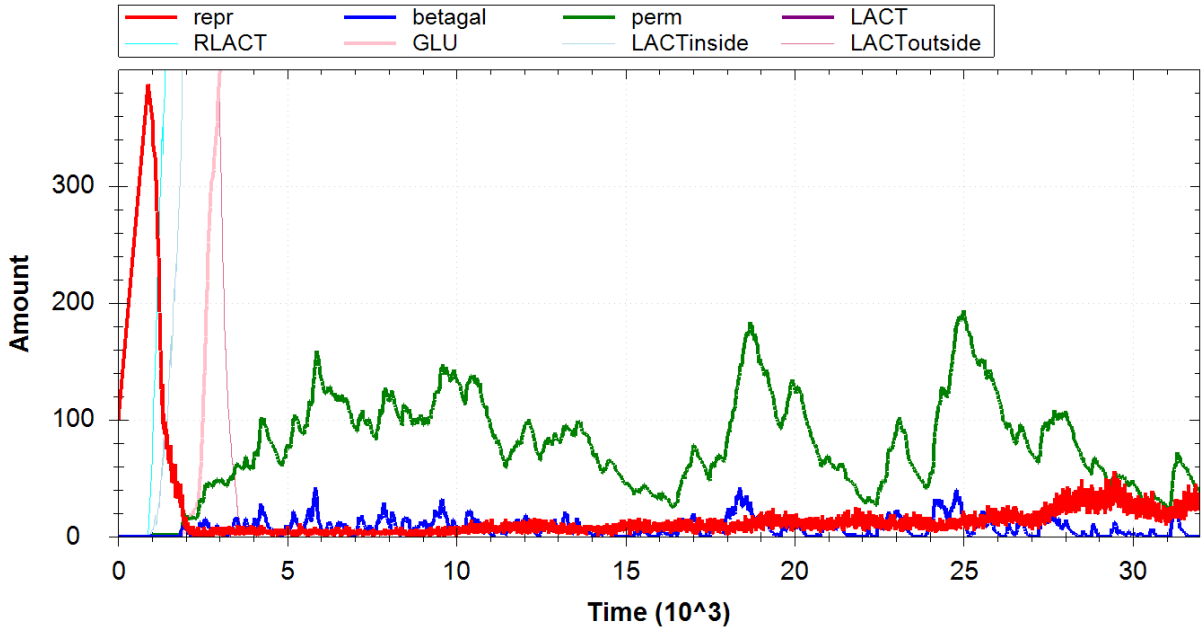


Figure 4.10: Zoom on the results of simulation of the regulation process of lactose operon in *Escherichia coli* when lactose is present in the environment.

4.5 Quorum Sensing in *Pseudomas aeruginosa*

Traditionally, bacteria have been studied as independent individuals. Now, it is recognized that many bacteria have the ability of monitoring their population density and modulating their gene expressions according to this density. This process is called *quorum sensing*. The process of quorum sensing consists in two activities, one involving one or more diffusible small molecules (called *autoinducers*) and the other involving one or more transcriptional activator proteins (*R-proteins*) located within the cell. The autoinducer can cross the cellular membrane, and thus it can diffuse either out or in bacteria.

The production of the autoinducer is regulated by the R-protein. The R-protein by itself is not active without the corresponding autoinducer. The autoinducer molecule can bind to the R-protein to form an *autoinducer/R-protein complex*, which binds to a target of the DNA sequence enhancing the transcription of specific genes. Usually, these genes regulate both the production of specific behavioral traits (as we will show in the following) and the production of the autoinducer and of the R-protein.

At low cell density, the autoinducer is synthesized at basal levels and diffuse in the environment where it is diluted. With high cell density both the extracellular and intracellular concentrations of the autoinducer increase until they reach thresholds beyond which the autoinducer is produced autocatalytically. The autocatalytic production results in a dramatic increase of product concentration. Quorum sensing behavior is very widespread in bacteria. An example is given by the bacterium

Pseudomonas aeruginosa, a prevalent human pathogen [33]. The ability of P. aeruginosa to infect a host mainly is based on controlling its virulence by quorum sensing. The level of virulence expressed by isolated bacteria is very low, thus avoiding host response. When a colony has reached a certain density, the production of virulence factors is autoinduced by quorum sensing, and it is generally sufficient to overcome the defenses of the host. The quorum sensing system of P. aeruginosa has two regulatory systems. Here we are interested in the one regulating the expression of *elastase LasB*, named the *las system*. The two enzymes, *LasB elastase* and *LasA elastase*, are responsible for pulmonary hemorrhages associated with P. aeruginosa infections.

A schematic description of the las system is as follows:

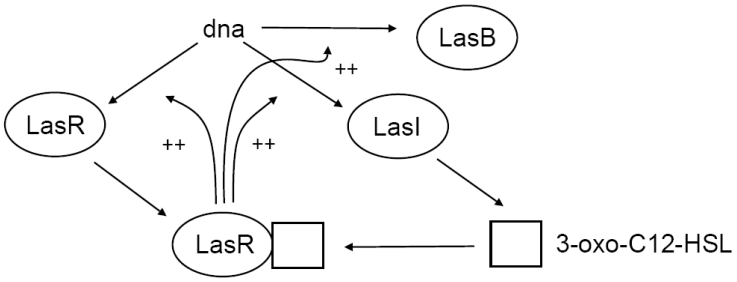


Figure 4.11: Schematic description of the las system in *Pseudomas aeruginosa*.

The autoinducer *3-oxo-C12-HSL* and the transcriptional activator protein *LasR* are produced at basal rates. The LasR/3-oxo-C12-HSL dimer is the activated form of *LasR*. It promotes the production of itself, of the autoinducer and of the *LasB* enzyme. The formation of the dimer is controlled mainly by the concentration of the autoinducer, which is influenced by the number of bacteria.

Stochastich CLS Model Quorum sensing is a complex biological process, which is not based on signals and receptors but only on concentration of a protein freely crossing membranes of bacteria. Many mathematical models have been developed for describing this challenging phenomenon [37, 55, 103]. These models consider various aspects of the problem: the diffusion of the autoinducer, its degradation, the percentage of up-regulated bacteria (the ones with an enhanced production of the autoinducer), the density of bacteria and their size, etc. However, all the models describe the process at a very abstract level. They consider that the intracellular concentration of the autoinducer is a function of the density of the bacteria, although modulated by other factors. Thus they start from this assumption to study the behavior of the system with different values of parameters.

The SCLS model is based on a different approach. A single bacterium is described by means of a set of rewrite rules modeling its internal processes. Such rules describe also that the autoinducer can cross (in both directions) the cellular membranes and that the autoinducer degrades at the same rate both inside and outside cells. Differently from the mathematical models mentioned above, the SCLS model describes the elementary processes each bacterium performs, and the quorum sensing results from the activity of a sufficient number of bacteria.

We now give the SCLS model of the quorum sensing process. We do not model the production of the *LasB* as it has not an active role in the regulation process. The initial state of each bacterium is:

$$Bact \quad ::= \quad \big(m\big)^L \rfloor (lasO.lasR.lasI)$$

where the looping sequence $\big(m\big)^L$ represents the bacterium membrane, *lasO* the target of the DNA sequence where LasR/3-oxo- C12-HSL complex binds to for promoting DNA transcription, and *lasR* and lasI the genes that encode *LasR* and the autoinducer.

This model shows one of the advantages of using terms for describing the structure of biological systems in SCLS. In fact, in order to model a population of n bacteria we have to describe only one bacterium, and then compose n copies of such a description by using the parallel composition operator. In other words, we model a population of n bacteria simply as $n \times Bact$.

The rewrite rule schemata describing the protein/protein and protein/DNA interactions in the described systems are now given. Again, we have only to give the rules for one bacterium, and they will be applicable in all the n bacteria of the considered population.

$$lasO.lasR.lasI \overset{20}{\mapsto} lasO.lasR.lasI \,|\, LasR \tag{S1}$$

$$lasO.lasR.lasI \overset{5}{\mapsto} lasO.lasR.lasI \,|\, LasI \tag{S2}$$

$$LasI \overset{8}{\mapsto} LasI \,|\, 3oxo \tag{S3}$$

$$3oxo \,|\, LasR \overset{0.25}{\mapsto} 3R \tag{S4}$$

$$3R \overset{400}{\mapsto} 3oxo \,|\, LasR \tag{S5}$$

$$3R \,|\, lasO.lasR.lasI \overset{0.25}{\mapsto} 3RO.lasR.lasI \tag{S6}$$

$$3RO.lasR.lasI \overset{10}{\mapsto} 3R \,|\, lasO.lasR.lasI \tag{S7}$$

$$lasO.lasR.lasI \overset{1200}{\mapsto} lasO.lasR.lasI \,|\, LasI \tag{S8}$$

$$lasO.lasR.lasI \overset{300}{\mapsto} lasO.lasR.lasI \,|\, LasR \tag{S9}$$

$$\big(m\big)^L \rfloor 3oxo \,|\, X \overset{30}{\mapsto} 3oxo \,|\, \big(m\big)^L \rfloor X \tag{S10}$$

$$3oxo\big(m\big)^L \rfloor X \overset{1}{\mapsto} \big(m\big)^L \rfloor 3oxo \,|\, X \tag{S11}$$

$$LasI \overset{1}{\mapsto} \varepsilon \tag{S12}$$

$$LasR \overset{1}{\mapsto} \varepsilon \tag{S13}$$

$$3oxo \overset{1}{\mapsto} \varepsilon \tag{S14}$$

Schemata (S1) and (S2) describe the production from the DNA of proteins LasR and LasI, respectively. For the sake of simplicity we do not model the transcription of the DNA into mRNA. Schema (S3) describes the production of the autoinducer 3-oxo-C12-HSL, denoted 3oxo, performed by the LasI enzyme. Schemata (S4) and (S5) describe the complexation and decomplexation of the autoinducer and the LasR

protein, where the complex is denoted 3R. Schemata from (S7) to (S9) describe the binding of the activated autoinducer to the DNA and its influence in the production of LasR and LasI. Schemata (S10) and (S11) describe the autoinducer exiting and entering the bacterium. The kinetic constants associated with these two schemata give a measure of the autoinducer dilution. Finally, schemata from (S12) to (S14) describe the degradation of proteins.

Simulation Results We simulated the behavior of a population of P. aeruginosa by varying the number of individuals. In Figure from 4.12 to 4.15 we show how the concentration of the autoinducer varies inside bacteria when the population is composed by one, five and twenty and hundreds individuals. We show the autoinducer concentration inside one only bacterium (the concentrations inside the others are analogous)[3]. When the number of bacteria increases also the concentration of the autoinducer in the extracellular space increases. As a consequence the concentration of the autoinducer in the intracellular spaces increases as well and the quorum sensing process starts. Note that the kinetic constants of rule schemata (S10) and (S11) regulating the autoinducer exiting and entering the membrane cause the bacteria to maintain the autoinducer production mostly at a basal rate when the population size is one or five. When the population size is twenty or more the quorum sensing starts after a few seconds thus causing a very high autocatalytic autoinducer production. As we show in Figure 4.14 and 4.15 the amount of autoinducer grow quickly to values of some hundreds. Increasing the ratio between kinetic constants of (S10) and (S11) would cause the quorum sensing to be triggered when the number of individuals is bigger. As we see comparing the Figure 4.14 and4.15 the presence of high amount of bacteria causes that the production is more quick (after twelve seconds the simulation with 100 bacteria exceed 300 units, amount the the simulation with 20 bacteria exceed after more than thirty seconds).

The stochastic nature of SCLS allows observing fluctuations of the autoinducer concentration which in the first two cases considered are not sufficient to trigger quorum sensing. Moreover, our model shows the discrete behavior of the binding between the autoinducer/R-protein complex and DNA.

[3]To monitor the amount of autoinducer in a single bacteria we have enriched one of these with a fictitious element that acts as marker. Practically we have used the pattern
`autoinducer_inside : (loop(m)[3oxo|$t:X | signal], # (occ(3oxo,X)) + 1 # ).`

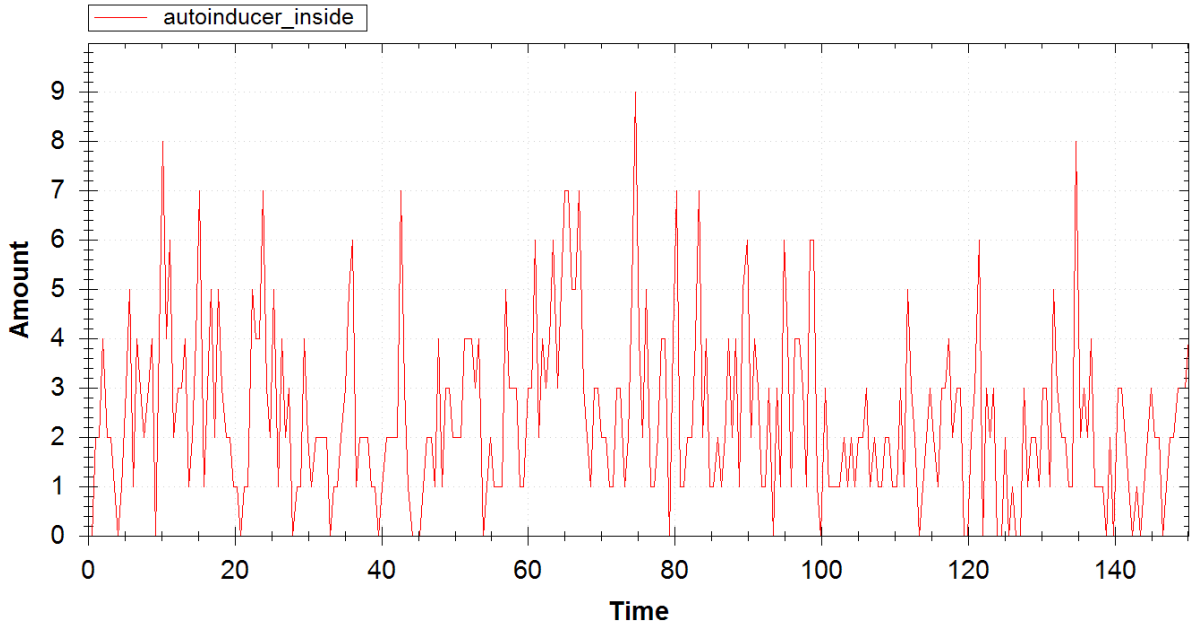


Figure 4.12: Result of simulation of quorum sensing in *Pseudomas aeruginosa*. Are shown the cases of **one** bacteria.

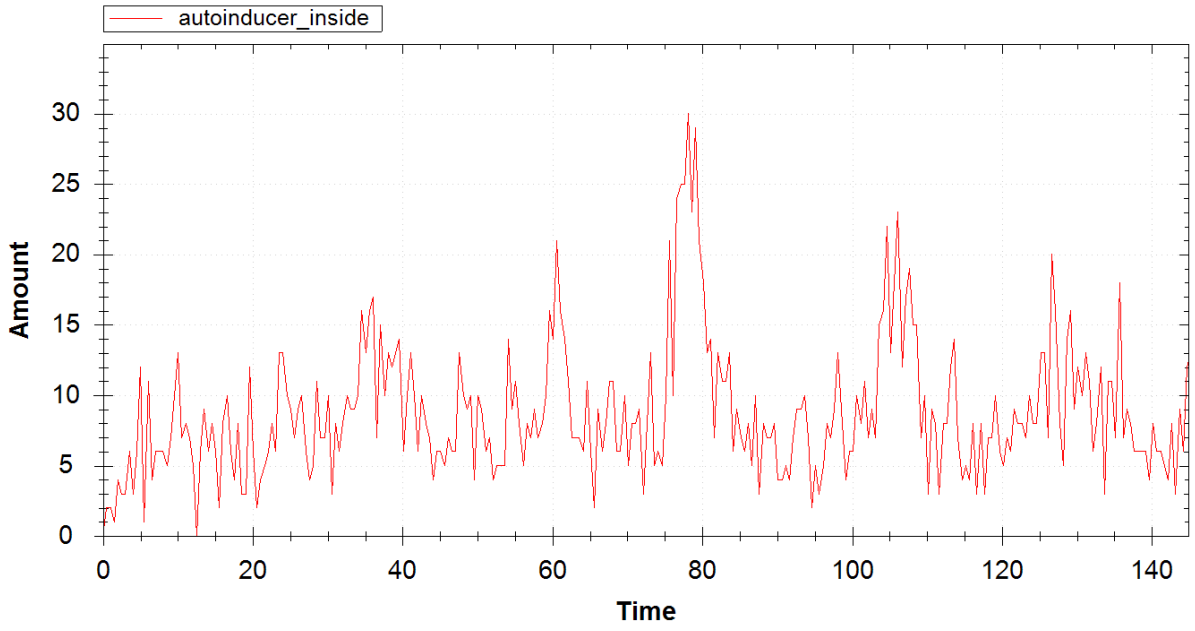


Figure 4.13: Result of simulation of quorum sensing in *Pseudomas aeruginosa*. Are shown the cases of **five** bacteria.

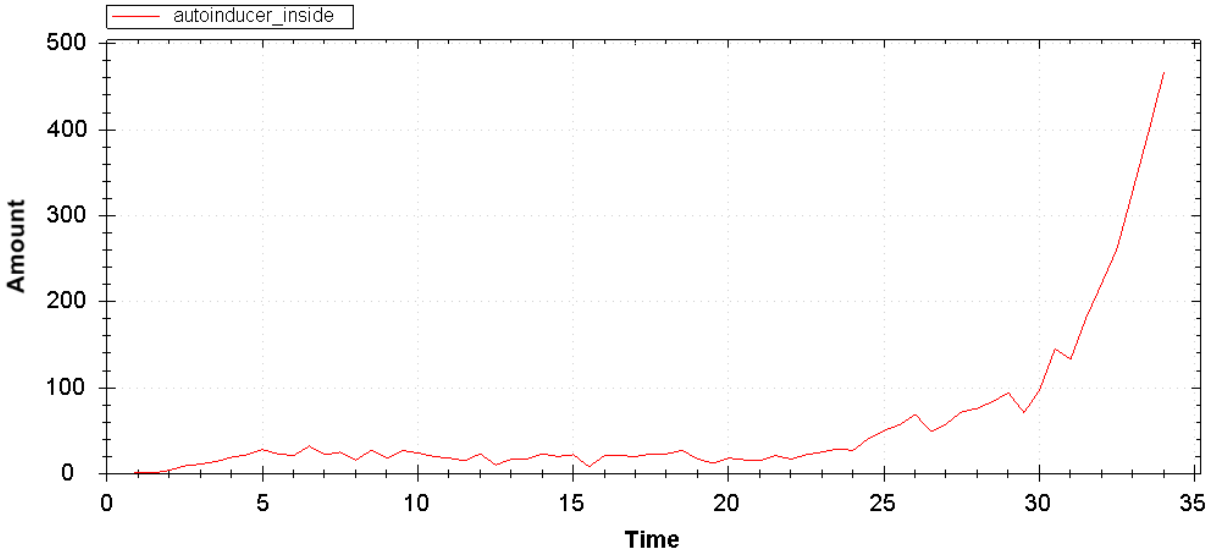


Figure 4.14: Result of simulation of quorum sensing in *Pseudomas aeruginosa*. Are shown the cases of **twenty** bacteria.

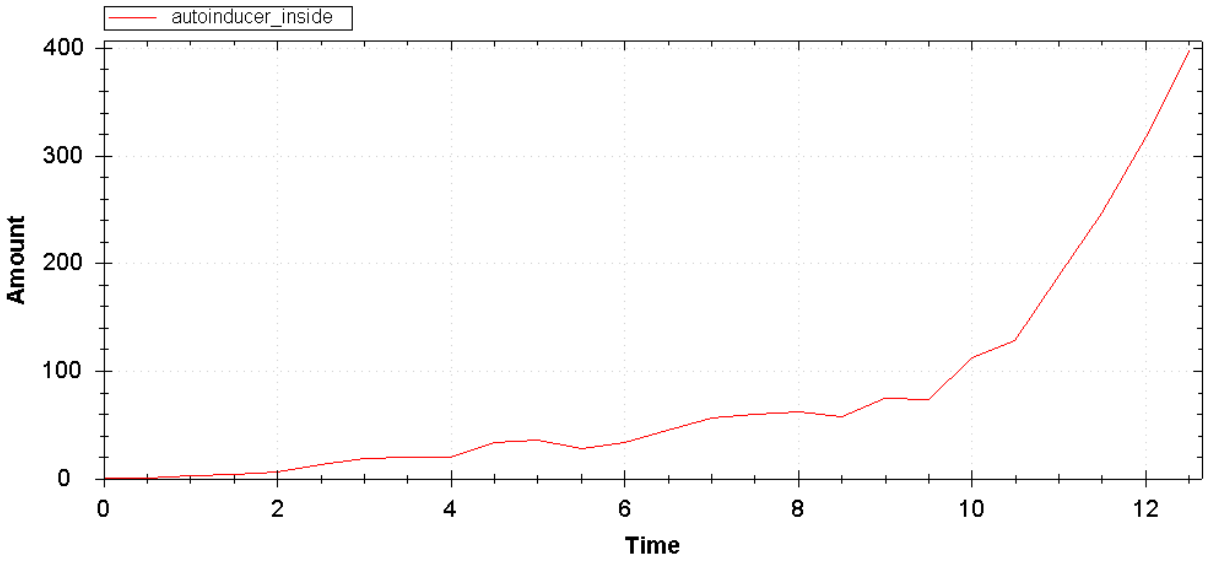


Figure 4.15: Result of simulation of quorum sensing in *Pseudomas aeruginosa*. Are shown the cases of one **hundred** bacteria.

4.5.1 Stiffness Evidence

In this use case it is evident the *stiffness*[75] problem of the Gillespie's SSA as we mentioned in Section 1.3.2. Whenever the complexity of the system increases, either through the increase of the number of possible reactions or through the a numerical

increase in the number of molecules in the system or an increase in the reaction rates, the algorithm adopts small $\Delta\tau$ in order to maintain the exactness of the simulation. In other word, the algorithm requires shorter timestep to capture the fast dynamic of the system. In evidence of this in Figure 4.16 we can see how while the simulation go forward, or better while the complexity of the system increase, the simulated time made by a single computation step decrease.

In Figure 4.17 we see that in the beginning of the simulation in the interval of one unit of simulated time we make 5 000 steps, whereas to simulate then in the same unit of simulated time we need to do 42 000 steps.

This phenomenon represent a problem regarding the time required for simulations. Unfortunately this problem is independent from how much the computation step is efficient and the only workaround is to use an enhanced SSA as discussed in Section 1.3.2.

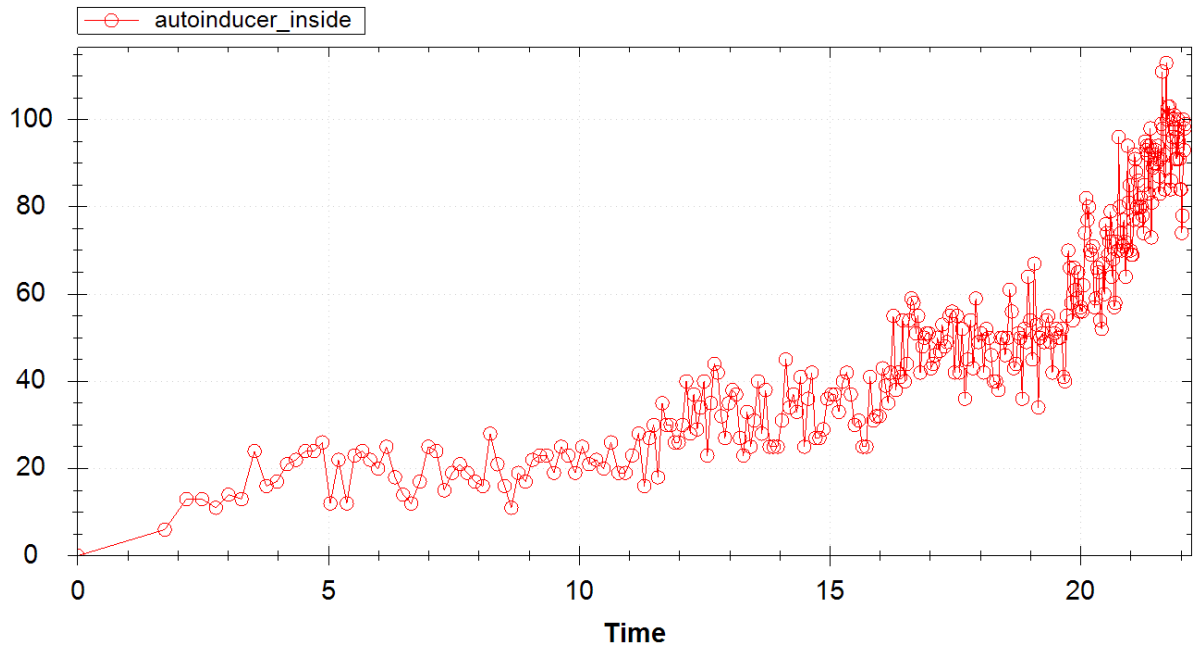


Figure 4.16: Evidence of *stiffness* in Gillespie's SSA in *Pseudomas aeruginosa* simulation. Here is shows how the simulation do steps more and more small as the simulation is left over. (in the graphs there is a point each 1000 iterations).

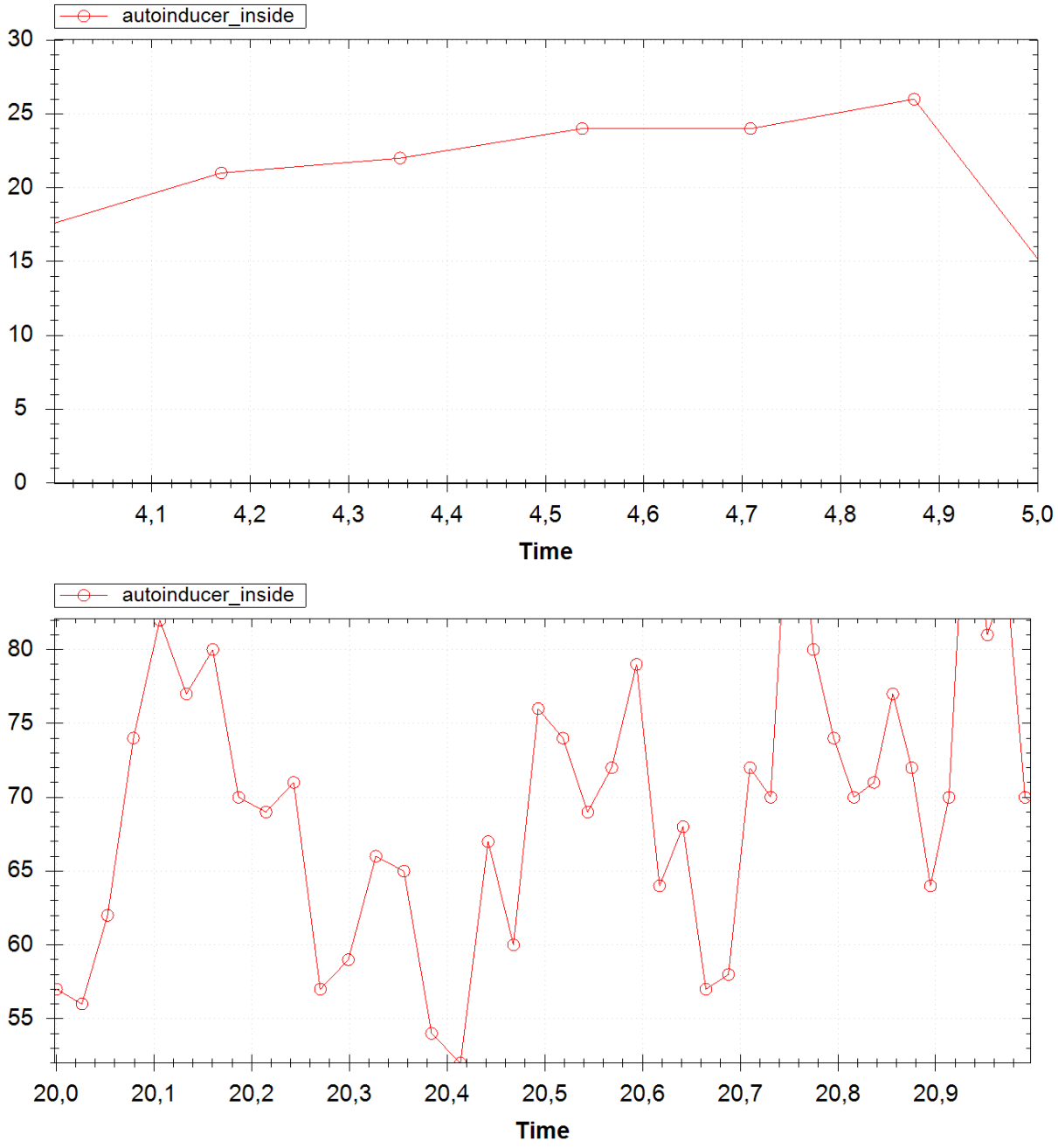


Figure 4.17: Evidence of *stiffness* in Gillespie's SSA in *Pseudomas aeruginosa* simulation. The two graph shows how the same simulated times requires a substantial different number of steps to be simulated as the simulation go forward. In the graph on the top in order to simulate 1 unit of time are necessary 5 000 steps whereas in the graph on the bottom to simulate the same simulated time we need to do 42 000 steps. (In the graphs there is a point each 1000 iterations.)

Chapter 5

Benchmarks

In this chapter we examine some benchmarks of simulation cases. Firstly we compare the performance of the developed algorithm (based on pre–processing and bottom–up match phase) whit the performance of the naive algorithm implemented in an early C++ prototype (5.0.2). Then we show some benchmarks that evidence the validity of the use of the *memoization* pattern (5.0.3) and of the optimized updating procedure whit *state–saving* feature (5.0.4).

All the test are executed on Windows XP operating system[1] on a AMD AthlonXP 2600+ @ 2000 MHz.

[1] Except for the comparison with C++ simulator. In fact the simulator runs only on unix systems and thus did the benchmark of both simulator on Ubuntu 7.04 on the same machine described.

5.0.2 Naive vs Pre–processing Algorithm Benchmarks

In Figure 5.1 we compare the performance of the C++ simulator, that use the naive algorithm for CLS pattern matching, with the performances of the simulator with the pre–processing algorithm with bottom–up matching (without the optimization of the updating procedure). The use case taken in exams is that of lactose operon in *Escerichia coli*[2] (see Section 4.4). As we can see that the use of the developed algorithm leads to an improve in performance of hundreds times.

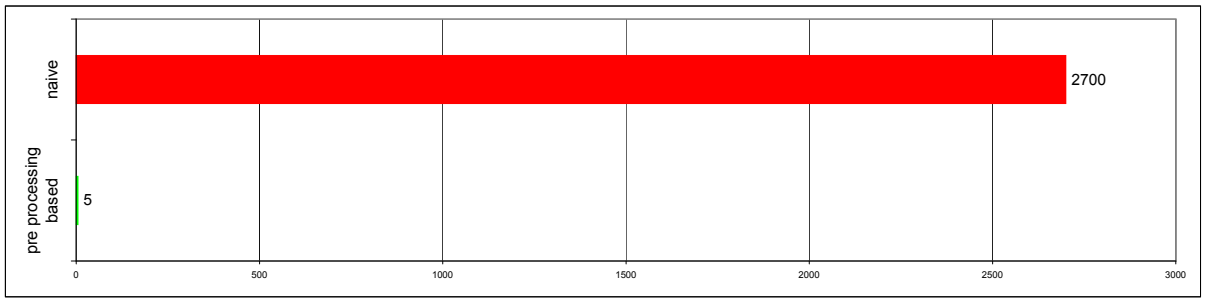
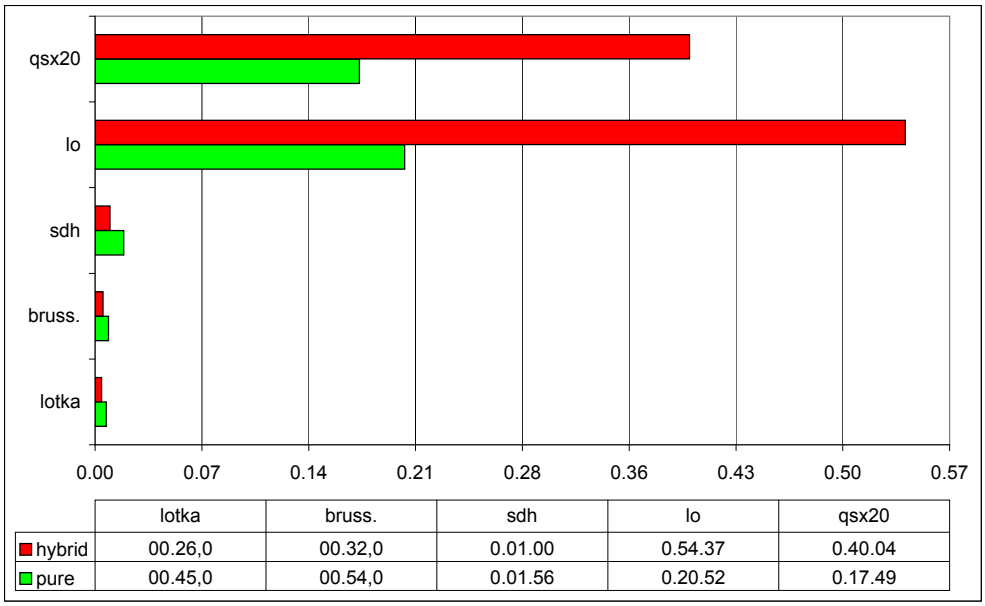


Figure 5.1: Benchmark result of simulation with the developed bottom–up pre-processing algorithm. The upper bar corresponds to the simulation with naive engine, whereas the lower indicates the running time of the developed algorithm (the time is expressed in seconds). Note that there are the times need to compute the first 1000 steps of the simulation and thus the manner times include the preprocessing time.

[2]Practically the lactose operon simulation has must be rewrote to adapt to the abilities of the prototype simulator. Then the same input are given to both simulators.

Note on the Performance of Chemical Simulations The performance of the simulator is not particular exciting in the cases of Lotka, Brussellator and of SDH; this is because this kind of simulation represent the worse case for own pattern matching algorithm. In fact, we work on a plain term and thus we do not take advantage of assumption on CLS trees. Moreover, in these cases, there are not variables, and thus our bottom–up simulation algorithm has too expensive overhead. In fact if we replace the preprocessing engine with an hybrid version of the algorithm, that uses the pre–processing algorithm only for the rules that contain variables, whereas uses a naive constant search for ground rules, we obtain performance enhancements, as we show in Figure 5.2. We show also how when the complexity of the term increases (as in lactose operon in e.c. and quorum sensing in p.a.) this is not true anymore.

For this reason we have included in the simulation engine a boolean flag that allows to use the pre–processing engine only for rule schemata, deals eventually ground rules with the naive constant search algorithm[3].



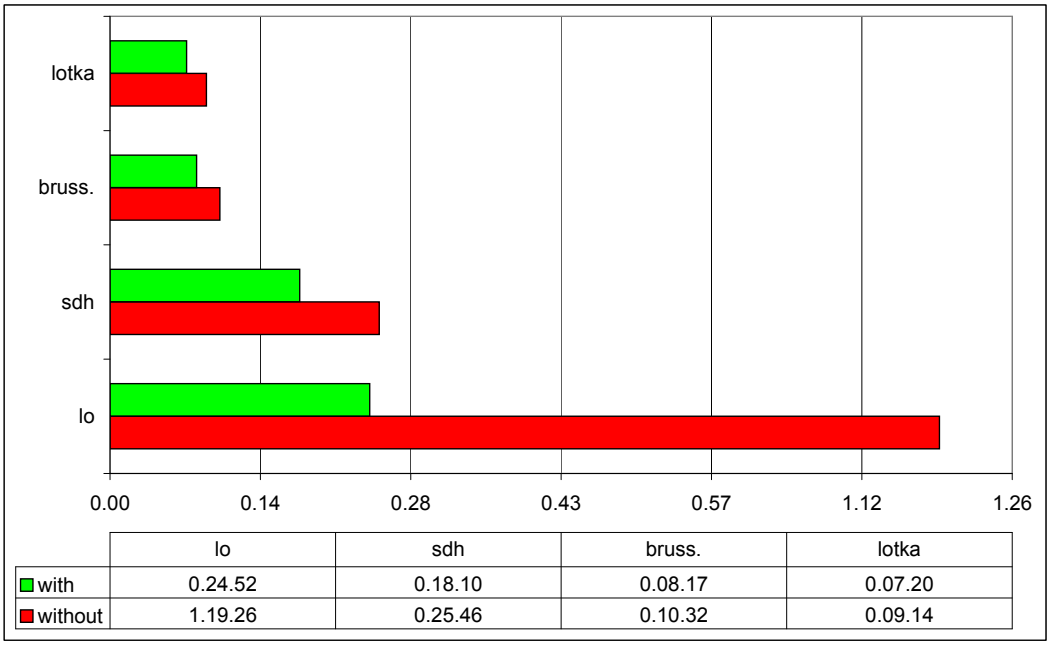
	lotka	bruss.	sdh	lo	qsx20
hybrid	00.26,0	00.32,0	0.01.00	0.54.37	0.40.04
pure	00.45,0	00.54,0	0.01.56	0.20.52	0.17.49

Figure 5.2: Benchmark result of hybrid vs pure pre–processing algorithm. The time is expressed in `hh.mm.ss` format. We can see that for plain term simulation, as Lotka, SDH and Brussellator, the naive algorithm with constant search improve the performance. We see also that if the complexity of the simulation term increase, as in the case of lactose operon and quorum sensing simulations this is not true anymore.

[3]this feature is selectable through option menu in the GUI of the developed simulator

5.0.3 Memoization Pattern Benchmarks

We have compared the performance of the simulation for 10^6 steps of simulation; as shown in Figure 5.3 the use of memoization pattern leads to a speed–up of up to about 2 times.



	lo	sdh	bruss.	lotka
with	0.24.52	0.18.10	0.08.17	0.07.20
without	1.19.26	0.25.46	0.10.32	0.09.14

Figure 5.3: Benchmark result of simulation whit use of *memoization pattern*. We have compared 10^6 iterations times. The time is expressed in `hh.mm.ss` format. In both the cases the optimized term updating procedure is not used.

5.0.4 Optimized Updating Procedure Benchmarks

Finally we have compared the performance enhancement given by the optimization of the updating procedure as we had see in Section 3.3.2. We can see how this approach allows to the algorithm to scale on great instances of input problem. In fact, as we see in Figure 5.4 and 5.5 , while the complexity of the simulated systems increases the performance gain obtained with the optimized updating procedure increases.

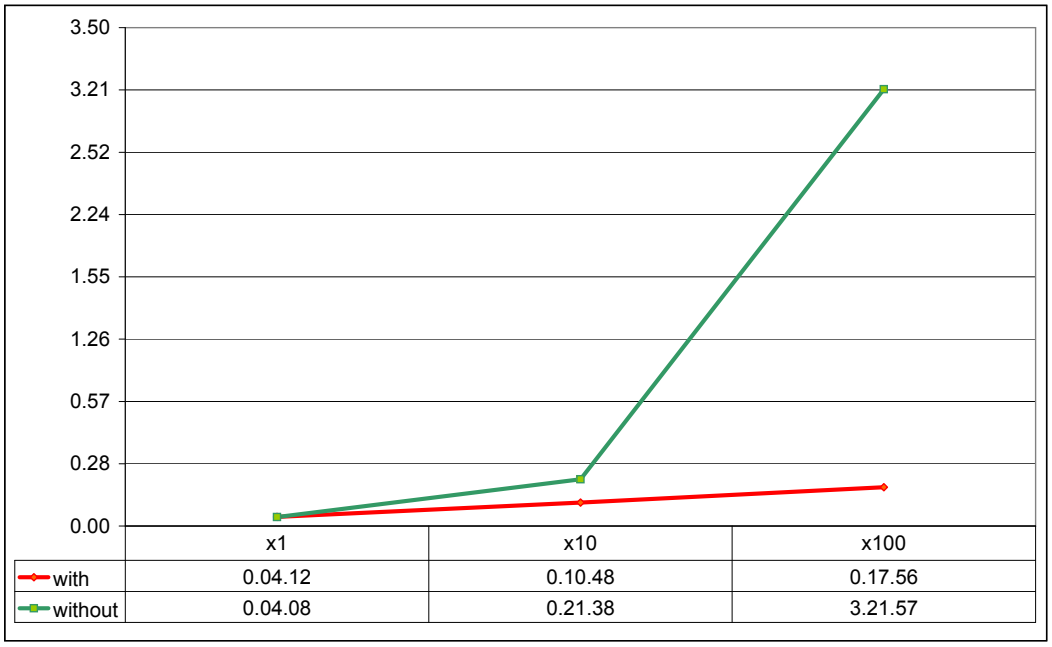


Figure 5.4: Benchmark result of simulation with use of optimized updating procedure. The use case analyzed is the simulation of quorum sensing in *Pseudomas aeruginosa*. We have measured the time elapsed varying the complexity of the simulated system; are shown the running time for one, ten and hundreds bacteria. The time is expressed in `hh.mm.ss` format. We compare the time elapsed for do 10^6 iteration by the algorithm without optimized updating procedure (green) against the performance of optimized one (red). We show how the performance of the optimized algorithm scale on bigger instances of the problem. A very similar result is obtained with the simulation case of genetic regulation process of lactose operon in *Escherichia coli*.

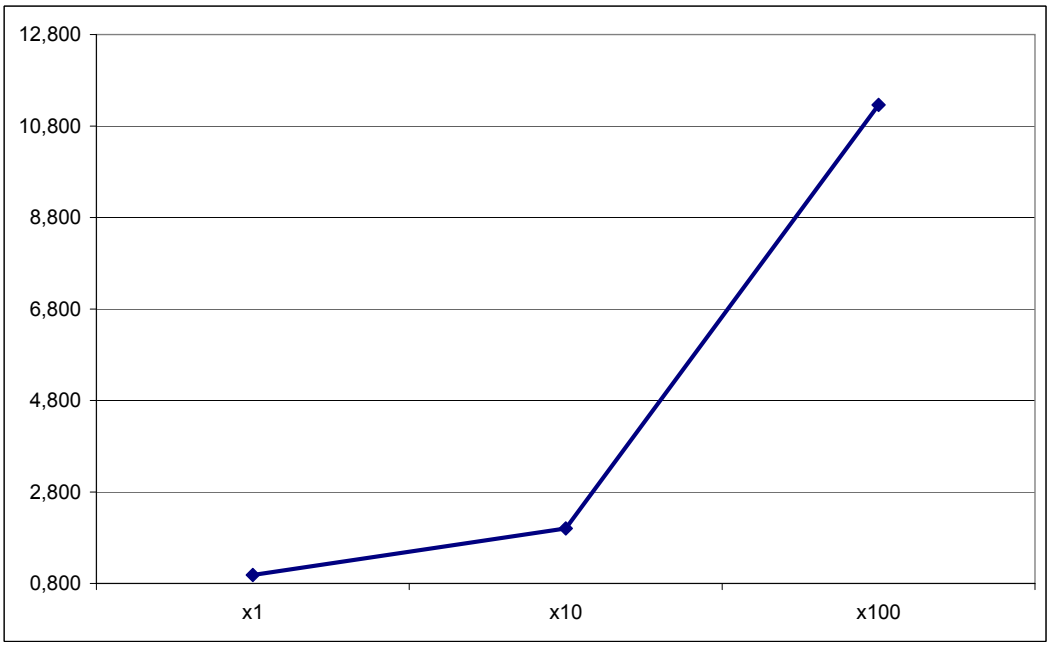


Figure 5.5: Benchmark of performance gain obtained with use of optimized updating procedure. We can see how the performance gain grows with the complexity of the simulated system. A very similar result is obtained with the simulation case of genetic regulation process of lactose operon in *Escherichia coli*.

Chapter 6

Conclusion

In this chapter we give the summary of the work of this thesis and then we analyze some possible future developments.

6.1 Summary

We have studied the problem of the implementation of a stochastic simulator for Calculus of Looping Sequences. We have extended the Gillespie's SSA to take account of rule schemata with rate functions, and we have designed and implemented an efficient algorithm for CLS pattern–matching. Moreover this algorithm has the vantage of being apt to optimizations that, allowing to preserve great part of matching information between successive steps of the computation, gives good response to local changes in state term (*state saving*), an thus allow to scale on complex simulation cases. The ideas proposed are shown to be valid by some benchmarks.

6.1.1 Software Development Details

The tools we have used to implement the simulator are

- Microsoft .NET Framework 2.0 / Mono 1.2.3 and

- FSharp Compiler v.1.9.2.9

We have developed a library contains all the data structure necessaries to represent an instance of SCLS simulation problem and the SCLS simulation engine (`scls.dll`).

We have produced about 5000 lines of F# code and about 1000 lines of C# code, that are well fused together thanks to the common intermediate language.

This development has required about 6 month of one man's work and has allow the experimentation of some original ideas.

This simulator is the first usable for SCLS allowing to express rule schemata.

6.2 Future Developments

The simulation machine should be formally specified, and the specification should be proved correct with respect to the calculus (like has been done for SPiM in [82]). More work remains, of course, both in the improvement of the performance of the simulation algorithm and in the user interface of the simulator.

6.2.1 Improvement of Performance

A first attempt to increase the performance can be obtained increasing the *node–sharing* in the pattern pre–processing procedures of the pattern matching algorithm: both among the NFA states and among the patterns of the rules. Moreover more work could be done to improve the representation of sequences; we can store these in a compressed way like we do for parallel composition nodes.

Other possible ways to improve performance are :

- *Using enhances SSA.* Like we have mentioned in Section 1.3.2 and shown in Section 4.5.1, the stochastic simulation through Gillespie's SSA suffers of intrinsic performance degradation (*stiffness*), independently from as it is implemented. In fact, as growing of the complexity of the simulated system, in order to maintain the exactness of the simulation the algorithm makes steps more and more small, causing that the number of steps (and thus the amount of "real" time) necessaries to simulate a unit of time increases.

  Thus to improve the performance in such cases it is necessary to change the simulation algorithm, using an enhanced SSA (see Section 1.3.2). Would be possible to use an approximated SSA or an hybrid engine (for example using hybrid stochastic–O.D.E.[1] engine [23]).

  Moreover it is possible to develop a software that selects the appropriate SSA engine according to the input problem or to the evolution of the simulation.

- *Specialization of the Interpreter.* Since the interpretation cycle of the simulation engine must run for many millions of times, it is possible to reduce the instruction executed by that cycle through the specialization of the interpreter: given a specified problem as input we can obtaining a sort of custom simulator and then run it (as discussed in [91]). In fact the elimination of a small percentage of executed instructions from this cycle can be decisive for simulations that must be run for days.

6.2.2 Supports Simulations of CLS Variants

The CLS formalism is presented in [77] with two variants: *CLS+*, *CLLS*.

- *CLS+* is an extension of CLS in which the looping operator can be applied to parallel composition of sequences. This would allows modeling membranes

[1] Ordinary Differential Equations

in more natural way, even if this will requires the definition of more complex semantics[2].

In [77] is showed that, if it is valid a simple restriction on variables of the rules, it is possible to translate CLS+ models in CLS, by a pre–processing procedure, preserving the semantics of the model. Thus we could develop a plug–in for the simulator that do this translation and then execute the simulation with the standard SCLS engine.

- In [77] the CLS formalism is extended to represents protein interaction at the domain level. Such an extension, called *Calculus of Linked Looping Sequences* (CLLS), is obtained by labellings elementary components of sequences. Two elements with the same label are considered to be linked.

  Even if the introduction of labels requires the definition of a complex type system that guarantees the well–formedness of the CLLS terms and patterns (propriety that must be checked at each runtime computational step), in [77] is shown that if we verify the well–formedness of the term representing the starting state of the simulation, and we verify a simple restriction on the rules of a simulation case (called *compartment safety*), we have the guarantee that the system will evolves in well–formed states. In fact the application of a well–formed rule satisfying compartment safety to a well–formed term preserves the well–formedness of the term.

  Thus in order to extend the SCLS simulation to support the CLLS we should:

  – expand the CLS data structures representing the elements with labels as a syntactical informations;

  – develop the pre–processing procedures charged to verify the well–formedness of the starting term and of the set of rules.

6.2.3 Improvement of the User Interface

Firstly can offer to the user a more rich representation of the term evolution. In fact the plot of concentrations is a representation with loss of information: the underlined formalism deals with hierarchy of terms, that are lost in the plot of concentration. Other suitable representations could show the evolution of nested membranes, allowing for example to build some animations of the evolution of the simulation. Another simple enhancement could allows the user to make more complex realtime plots, as for example phase spaces. Finally a trivial extension of the interface of the simulation can be the design of an interface that allow the user to requires the execution of a queue of batch simulations (maybe with the possibility of state saving by serialization).

Interface for reverse Engineering A more complex interface could allows the user to discover some unknown constants using the simulator as reverse engineering tool. As example the user could wants to simulate a network of reactions of which

[2]Moreover this formalism does not suffer the ambiguity of the value of rate function as we see in Section 3.3.3 and in the Example 3.7.

not all the kinetic constants are known. Here the software could helps offering the way to simulate the system with some default constants; then the user can see the output plot and, interacting with it, express the expected behavior, known as response of the entire system from *in vitro* experiments. Then the simulator can run a battery of test varying the searched constants, in order to obtains the required behavior. Once obtained the expected output response, we have discovered the not known constants, without the necessity to measure these *in vitro*. Obliviously to make a similar tool we need a very efficient simulator and the help of some artificial intelligence algorithms.

MIMs Graphical Interface One of the discriminating factors for the success of a simulation instrument is that it is easily usable for the users interested in the systems that it simulates. To let this happens, it is necessary that the simulator presents an interface with a formalism that is familiar to the customers. In the case of CLS, the final user of the simulator will be biologists and chemists, and it is absurd to think that they can express their self directly in a formalism based on term rewriting. This problem is actually diffused in all inter–disciplinary withins; it is necessary, even if much laborious, that persons coming from different academic cultures succeed to find a common language that allows to understand their mutual requirements. Like we have seen in Section 1.1.2, one of the most well designed and rigidly defined proposals of graphical language are Kohn's Molecular Interaction Maps (MIMs) [11, 64, 66].

Since in the work of Milazzo is presented a way to encode Kohn MIM, a possible enhancement of the user interface of the simulator should be the development of an interface based on this graphical formalism. In this way the users could express the SCLS model in a more natural way.

This can be accomplished implementing MIM thought *Windows Forms Custom Controls*, building a control for each basic elements of MIM (see Figure 6.2). Given such Controls we could implement a translator from such diagram to SCLS models, like specified in [77].

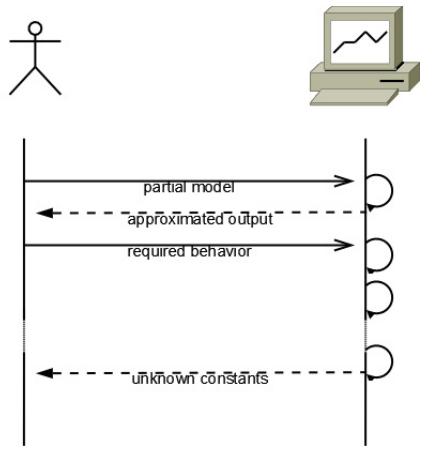


Figure 6.1: Interaction diagram of the use of the simulator as reverse engineering tool.

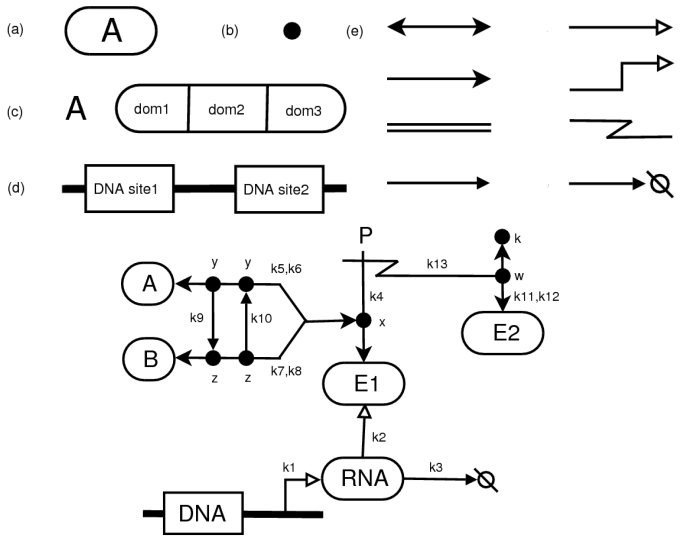


Figure 6.2: Example of MIM diagrams. Species in a MIM are as shown in case from (a) to (d), and reactions are depicted as arrows like in (e). On the bottom we can see a complete example of MIM diagram.

Appendix A

User Manual

A.1 License

All the software developed for this thesis is published at
http://www.di.unipi.it/~milazzo/biosims/ and is released under GNU G.P.L. [3].

A.2 System Requirements

- .Net 2.0 Platform. (tested on Microsoft .Net Framework 2.0 and on Mono 1.2.3)
- (F# library 1.9.2.9 installed in the GAC for the non standalone version)

A.3 Usage

A.3.1 Format of the Input file

SCLSm accepts text input file with the following sections. See A.1 for details.

Rules Here you can express the rules that controll the evolution of the system. This section must begin with "`rules`". The syntax to express each rules is the following:

$$nameoftherule : (lefthandside, righthandside, ratefunction)$$

where

- the name of the rule can be an alpha numeric identifier
- the left hand side and the right hand side are CLS term in which be variables. The syntax for express variable is

$$\$symbol : identifier$$

where symbol is "`e`" for element variables, "`s`" for sequence variables, "`t`" for term variables.

- the rate function can be either a constant or a piece of $C\#$ code delimited by a couple of # characters. Here the user can call arbitrary function, including user defined procedure. As example we had defined the `occ` function that count the occurrences of a set of constant elements inside some variable's instantiation. The syntax for this function is

  $occ(space\ separated\ constant\ identifiers, space\ separated\ variable\ identifiers)$

  For example `occ(A,X)` counts the occurrences of `A` in the instantiation of `X`, `occ(A B,X Y)` counts the summation of occurrences of `A` in `X`, of `B` in `X`, of `A` in `Y`, of `B` in `Y`.

Term Here you can express the initial configuration of the simulation. This section must begin with "`term`". The syntax to express term are the following

- Each element it is expressed by an alpha numeric identifier.
- Sequence are builds of element separated by a "`.`"
- Compartments are builds of sequences and loop separated by a "`|`". Each element constituent of a compartment can be followed by an "`*`" and by an integer value that indicates the number of repetition of the element, in absence of which it is assumed as one repetition.
- Loop are builds of a sequence (looping) representing the membrane and a compartment representing the content. The sintax are

$$loop(membrane)[content]$$

User's Patterns User's pattern are intended to allow the user to express patterns that want to monitor in the plot and in the writing of outputs. This section must begin with "`patterns`". The syntax for express these pattern is

$$identifier \quad : (patter\ ,\ constant\ or\ rate\ function\ )$$

The constant or the rate function will be used to multiply he number of occurrences of the pattern. This is often useful as in the following example : `LACTinside: (loop($s:x)[$t:X | LACT], # occ(LACT, X) + 1 # )`. This will count the number of LACT inside of a membrane. Something, when we have multiple instances of certain cell, is also useful to monitor some concentrations that regards only a single cell. This is possible adding a fictitious element inside the interested cell as in the example of quorum sensing attached with the software distribution.

Exclude list Here user can express what elements want to exclude from plotting. This section must begin with "`exclude`". The syntax to express these patterns is

$$identifier|ALL - identifier$$

It is possible to tell what identifier evolution want not to be monitored or to tell that want to exclude ALL identifier except the identifier list that follow the "`-`".

Note that the time scale depends on the constants of the rules; thus if is required a plot with time scale in seconds, the constants must be appropriately scaled.

A.3.2 Description of the User Interface

Main Window

The main features of the graphical user interface of the simulation are show in Figure A.1. We now examine each of the GUI's components.

Menu Through menus is possible to open an input file and start the simulation or quit the application (File); to view the input file (View), to start, stop or pause the simulation, to save the simulation's evolution concentration to text, html or spreadsheet, to update graph and to hide/show the graph legend (Actions); to switch between time and iteration limit mode (Options) an to view help or about (Help). Note that in the iteration limited simulations the sampling rate is expressed in iteration (will sample the amounts each n iteration) whereas in the time limited simulations the sampling rate is expressed in milliseconds (will sample the amounts each n / 1000 time units).

Parameters Controls Through the parameters controls is possible to select the input file describing the simulation instance (and start the simulation) and to set the limit and the sample rate of the concentrations. In the case of timed limited simulation we can give a stop time and each much milliseconds we want to sample the concentrations in the simulation's state. In case of iteration limit we can choose after how much iteration the simulator will stop and every how much iterations the concentrations are measured.

Graph Panel In the graph panel we will get the plot of the concentration's evolution. The curves that will plot are specified in the input file through the `EXCLUDE` field. Through the interaction with this panel we can change the color, the thickness or the symbol of the curves, zoom or pan the plot as show in Figure A.1. Clicking on the left mouse's button it is shown a context menu through which it is possible to save the plot as an image, ask to show the point's value and other options.

State Informations When a simulation is started we can see in the bottom of the simulation's state: the status bar show the percentage of simulation that is done. Close to the status bar there are the indication of number of iterations and time elapsed by the simulation.

Input File View Windows

This windows, enabled through Option menu, show the input file with highlighted keywords.

Term Tree View Windows

This window is shown when we pause or stop the simulation. We can see the current state's term in a tree view. Click on a entry with right mouse's button we can expand or collapse the entry; by left mouse we can expands all the entries.

A.4 Known limitations

A.4.1 Multiple Term Variables not on the Top Level of Compartments

Currently the simulator allows to do simulations with rule with multiple term variable if there are at the top–level of the left hand side of a rule. In details it allows to use any number of term variable inside a rule, but, except the case in which the variables are located in top–level of a rule, there is the following limitation: there must not be more than one variable for level.

As example the term
$X \,|\, Y \left(m\right)^L \rfloor \, X \,|\, \left(\widetilde{x}\right)^L \rfloor \, (Y \,|\, a.b \,|\, \left(\widetilde{y}.\widetilde{x}.\widetilde{z}\right)^L \rfloor \, (a \,|\, X)))$ is a valid left hand side of a rule because the only case in which are introduced more than one variable for level are in the top–level of the rules. Whereas
$\left(m\right)^L \rfloor \, (X \,|\, Y \,|\, a)$ is not a valid left hand side of a rule because there are more than one term variables introduced for level.

We have left this limitation because this is a tread–off between complexity of the bottom–up inference engine and the allowed expressiveness of the rules; we not have found any biological significant use cases that requires this feature.

In the source code this cases give a `Not Yet Implemented` exception[1]

A.4.2 Number of Occurrences of Reactants

There is the limitation of using a 64bit integer for representing the number of occurrence of a reaction in a solution. That is the number of that occurrence must be represented in 64bit space. If the number of times in which the reactants of a rule exceed that limit we will get an overflow and thus to a not correct behavior.

A.5 Guide to Released Source Files

SCLS folder

- `scls.fsi` and `scls.fs` contains the definition of the data structures of the application's domain. Namely here we find :
- `nfa_leaves_matcher.fsi` and `nfa_leaves_matcher.fs` contains the definition of the non deterministic finite state automaton that is the responsible of compute the match of a leaf (= a ground CLS sequence or looping sequence) against a set of sequence patterns
- `preprocessing.fsi` and `preprocessing.fs` contains the definition the data structures necessaries to the pre–processing engine.
- `preprocessing_engine.fsi` and `preprocessing_engine.fs` contains the engine of the simulation; more in details the engine that realy on the pre–processing of data structures for the match phase. Moreover here is defined the extension of Gillespie's SSA that take in account of rule schemata with variables and rate functions.

[1] In the source this limitation is located in `preprocessing.fs` file

SCLS_Parser folder contains the definition of the parser of SCLS input file as defined in Section A.3.2. This folder contains also the SCSL.atg attributated grammar used by cocoR(see http://www.scifac.ru.ac.za/coco/) for the scanner and parser generation.

SCLS_Simulator folder contains the definition of the worker and client of the interface to simulation engine.

CodeExpressionEvaluator folder contains the definition of a class that compile and load dynamically some given C# code.

Listing A.1: Syntax of input files in BNF.

```
CHARACTERS
        DIGIT =         "0123456789".
        LETTER =        "ABCDEFGHILMNOPQRSTUVZWXYKJabcdefgihlmnopqrstuvzwyxkj".
        SYMB =          "+-/ !%&=?^     _".

TOKENS
        Id = LETTER [{LETTER | DIGIT | SYMB}].
        Number = DIGIT [{DIGIT}] ['.' DIGIT [{DIGIT}] ].

        VART =          "$t:".  //      term variable dichiaration
        VARS =          "$s:".  //      sequence variable dichiaration
        VARE =          "$e:".  //      element variable dichiaration

        RULESDIC =      "rules" .
        TERMDIC =       "term" .
        PATTERNDIC = "patterns" .
        EXCLUDEDIC = "exclude" .
        CODEDELIMITER = "#" .

COMMENTS FROM "/*" TO "*/" NESTED
COMMENTS FROM "//" TO '\n'
IGNORE '\n' + '\r' + '\t'

PRODUCTIONS

SCLS
                =
                        RULESDIC rules
                        TERMDIC terms
                        [PATTERNDIC [patterns]  ]
                        [EXCLUDEDIC [ [Identifier] { Identifier } | "ALL" ["-" Identifier
                             { Identifier } ] ] ]
                .
rules
                =
                        rule    [ rules ]
                .
rule
                =
                        Identifier
                        ':' '(' terms   ',' [terms] ','
                             rateFormula    ')'
                .

terms
                =
                        term    ['*' Number] [  '|' terms       ]
                .
term
                =
                        sequences
                |       VART Identifier
                |       ["loop"] '(' [sequences]  ')'[ enclosedTerm ]
                .
enclosedTerm
                =       '[' terms ']'
            .
sequences
                =
                sequence        ['.' sequences ]
                .
sequence
                =
                                Identifier
                |       VARE    Identifier
                |       VARS    Identifier
                .
patterns
                =
                pattern [ patterns ]
                .
pattern
                =
                Identifier      ':' '(' terms [',' rateFormula] ')'
                .
rateFormula
                =
                        CODEDELIMITER ANY       {       ANY             }  CODEDELIMITER
                |       ANY
                .
Identifier
                =       [Number ]       Id
                .
```

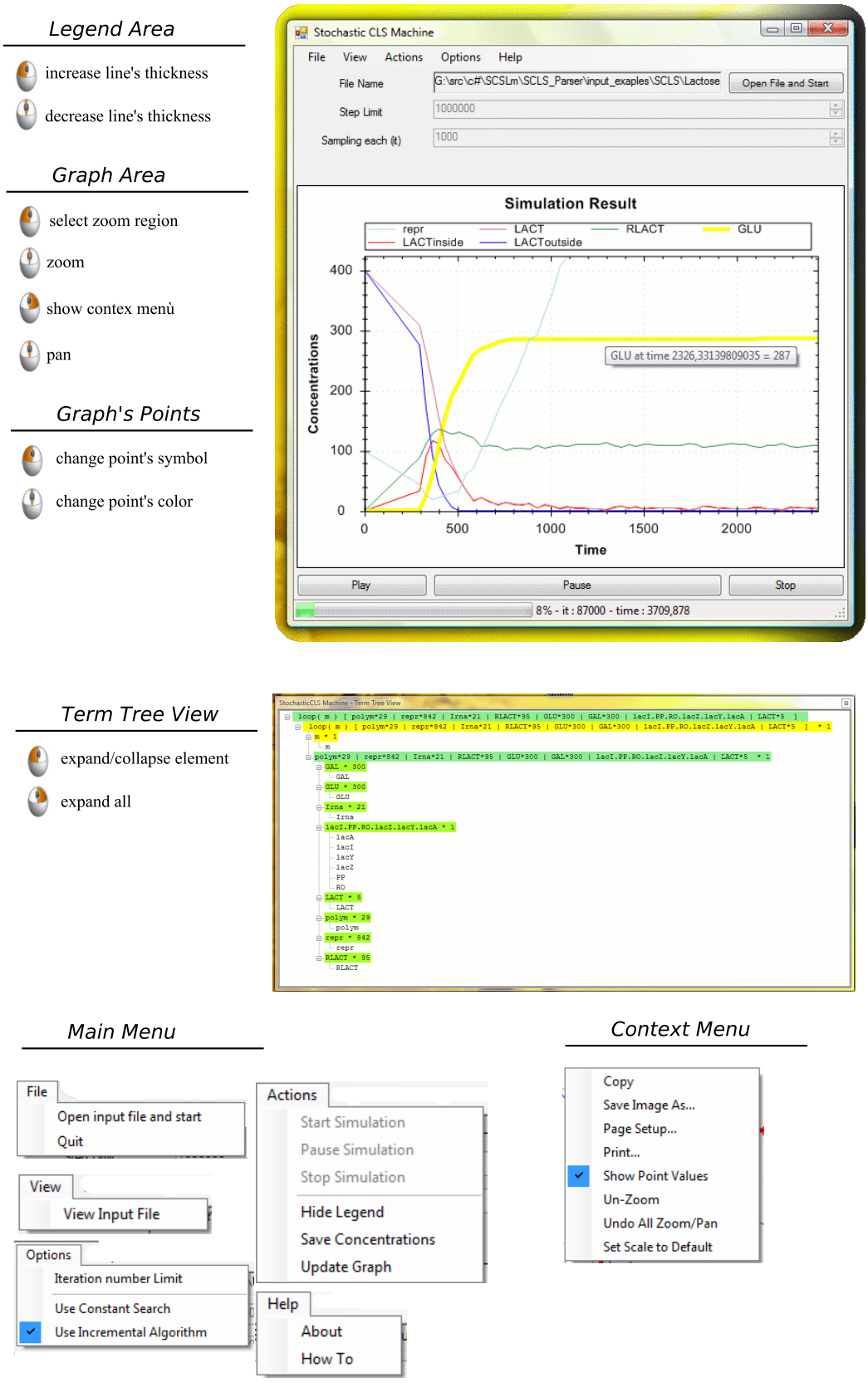


Figure A.1: Overview of the simulator user interface.

Listing A.2: Example input file for SCLSm (Quorum Sensing Example).

```
rules
R1:     ( lasO.lasR.lasI   , lasO.lasR.lasI|lasR+       , 20   )
R2:     ( lasO.lasR.lasI   , lasO.lasR.lasI|lasI+       , 5    )
R3:     ( lasI+            , lasI+ | 3oxo               , 8    )
R4:     ( lasR+ | 3oxo     , 3R                         , 0.25 )
R5:     ( 3R               , lasR+ | 3oxo               , 400  )
R6:     ( 3R|lasO.lasR.lasI , 3RO.lasR.lasI             , 0.25 )
R7:     ( 3RO.lasR.lasI     , R|lasO.lasR.lasI          , 10   )
R8:     ( 3RO.lasR.lasI     , 3RO.lasR.lasI|lasR+ , 1200   )
R9:     ( 3RO.lasR.lasI     , 3RO.lasR.lasI|lasI+ , 300    )
R10:    ( loop(m)[3oxo|$t:X] , 3oxo | loop(m)[$t:X] , # 30 * (occ(3oxo,X)) #)
R11:    ( 3oxo | loop(m)[$t:X], loop(m)[3oxo|$t:X] , 1 )
R14:    ( lasI+  ,  , 1 )
R15:    ( lasR+  ,  , 1 )
R16:    ( 3oxo   ,  , 1 )
term
loop(m)[lasO.lasR.lasI|signal] | loop(m)[lasO.lasR.lasI]*20
patterns
autoinducer_inside : (loop(m)[3oxo|$t:X|signal], # (occ(3oxo,X)) # )
exclude
ALL - m
```

Bibliography

[1] CellIllustrator. See http://www.cellillustrator.com/.

[2] CLIPS. See http://www.ghg.net/clips/CLIPS.html.

[3] GNU General Public Licence. See http://www.gnu.org/copyleft/gpl.html.

[4] MCell. See http://www.mcell.psc.edu/.

[5] P–System. See http://psystems.disco.unimib.it/.

[6] SWB (System Biology Workbench). See http://sbw.sourceforge.net/.

[7] The System Biology website. See http://www.systems-biology.org.

[8] VirtualCell. See http://www.ibiblio.org/virtualcell/index.htm.

[9] J.L. Abkowitz, S.N. Catlin, and P. Guttorp. Evidence that hematopoiesis may be stochastic in vivo. *Nature Medicine*, 2:190–197, 1996.

[10] A. V. Aho and M. J. Corasick. Efficient string matching: an aid to bibliographic search. *Commun. ACM*, 18(6):333–340, 1975.

[11] M. I. Aladjem, S. Pasa, S. Parodi, J. N. Weinstein, Y. Pommier, and K. W. Kohn. Molecular interaction maps–a diagrammatic graphical language for bioregulatory networks. *Science's STKE*, 2004(222):pe8, 2004.

[12] B. Aleman-Meza, H. B. Schuttler, J. Arnold, and T.R. Taha. Kinsolver: A simulator for biochemical and gene regulatory networks. 2002.
See http://webster.cs.uga.edu/~boanerg/mams.

[13] R. Alur, C. Belta, F. Ivancic, V. Kumar, M. Mintz, G.J. Pappas, H. Rubin, and J. Schug. Hybrid modeling and simulation of biomolecular networks. *LNCS*, 2034 - Hybrid Systems: Computation and Control:19–32, 2001.

[14] F. Baader and T. Nipkow. *Term Rewriting and All That.* Cambridge University Press, 1999.

[15] J. Backus. Can programming be liberated from the von neumann style?: a functional style and its algebra of programs. *Commun. ACM*, 21(8):613–641, 1978.

[16] R. Barbuti, S. Cataudella, A. Maggiolo-Schettini, P. Milazzo, and A. Troina. A probabilistic model for molecular systems. *Fundamenta Informaticae*, 67:13–27, 2005.

[17] R. Barbuti, A. Maggiolo-Schettini, P. Milazzo, P. Tiberi, and A. Troina. Stochastic cls for the modeling and simulation of biological systems. *Submitted to Bioinformatics*, 2007.

[18] R. Barbuti, A. Maggiolo-Schettini, P. Milazzo, and A. Troina. An alternative to gillespie's algorithm for simulating chemical reactions. *Computational Methods in Systems Biology (CMSB'05)*, 2005.

[19] R. Barbuti, A. Maggiolo-Schettini, P. Milazzo, and A. Troina. A calculus of looping sequences for modelling microbiological systems. *Fundamenta Informaticae*, 72:21–35, 2006.

[20] R. Barbuti, A. Maggiolo-Schettini, P. Milazzo, and A. Troina. Bisimulations in calculi modelling membranes. *Submitted to Formal Aspects of Computing*, 2007.

[21] R. Barbuti, A. Maggiolo-Schettini, P. Milazzo, and A. Troina. The calculus of looping sequences for modeling biological membranes. *LNCS*, to appear - 8th Workshop on Membrane Computing, 2007.

[22] W. J. Blake and J. J. Collins. And the noise played on: Stochastic gene expression and hiv-1 infection. *Cell*, 122, Issue 2:147–149, 2005.

[23] L. Bortolussi. *Constraint-based approaches to stochastic dynamics of biological systems*. PhD thesis, Department of Mathematics and Computer Science, University of Udine, Italy, 2007.

[24] R. S. Boyer and J. S. Moore. A fast string searching algorithm. *Commun. ACM*, 20(10):762–772, 1977.

[25] J. Cai, R. Paige, and R. Tarjan. More efficient bottom-up multi-pattern matching in trees. *Theor. Comput. Sci.*, 106(1):21–60, 1992.

[26] Y. Cao, D. T. Gillespie, and L. R. Petzold. Efficient step size selection for the tau-leaping simulation method. *The Journal of Chemical Physics*, 124, 2006.

[27] L. Cardelli. Abstract machines of systems biology. *Transactions on Computational Systems Biology*, III, LNBI 3737:145–168, 2005.

[28] L. Cardelli. Brane calculi. interactions of biological membranes. *LNCS*, 3082 - Computational Methods in Systems Biology (CMSB'04):257–280, 2005.

[29] D.R. Cox and H.D. Miller. The theory of stochastic processes. 1965.

[30] M. Curti, P. Degano, C. Priami, and C.T. Baldari. Modelling biochemical pathways through enhanced pi-calculus. *Theoretical Computer Science*, 325:11–140, 2004.

[31] Q. Dang and C. Frieden. New pc versions of the kinetic-simulation and fitting programs, kinsim and fitsim. *Trends Biochem. Sci.*, 22 (8):317, 1997.

[32] V. Danos and C. Laneve. Formal molecular biology. *Theoretical Computer Science*, 325(1):69–110, 2004.

[33] C. Van Delden and B. H. Iglewski. Cell-to-cell signaling and pseudomonas aeruginosa infections. *Emerg Infect Dis*, 4:551–560, 1998.

[34] N. Dershowitz and J. Jouannaud. *Rewrite Systems*, chapter 6, pages 243–320. J. van Leeuwen ed., 1990.

[35] P. Dhar. Cellware, 2003.
See http://www.bii.a-star.edu.sg/sbg/cellware.

[36] E.W. Dijkstra. E. W. Dijkstra manuscripts archive. EWD1036
See http://www.cs.utexas.edu/users/EWD/.

[37] J. D. Dockery and J. P. Keener. A mathematical model for quorum sensing in pseudomonas aeruginosa. *Bulletin of Mathematical Biology*, 63:95–116, 2001.

[38] R. B. Doorenbons. *Production Matching for Large Learning Systems*. PhD thesis, Computer Science Departement, Carnegie Mellon University, Pittsburg, PA, 1995.

[39] Robert B. Doorenbos. Production matching for large learning systems. Technical report, Pittsburgh, PA, USA, 2001.

[40] M. Ehlde and G. Zacchi. Mist: a user-friendly metabolic simulator. *Comput. Appl. Biosci.*, 11 (2):201–207, 1995.

[41] S. P. Ellner and J. Guckenheimer. *Dynamic Models in Biology*. Princeton University Press, 2006.

[42] C. Firth. *StochSim: a stochastic simulator of (bio)chemical reactions*. PhD thesis, University of Cambridge, 1998.

[43] C.L. Forgy. Rete: A fast algorithm for the many pattern/many object pattern matching problem. *Artificial Intelligence*, 19:17–37, 1982.

[44] D. T. Gillespie. Exact stochastic simulation of coupled chemical reactions. *Journal of Physical Chemistry*, 1977.

[45] D. T. Gillespie. The chemical langevin equation. *Journal Of Chemical Physics*, 113, 1:297–306, 2000.

[46] D. T. Gillespie. Approximate accelerated stochastic simulation of chemically reacting systems. *The Journal of Chemical Physics*, 115, issue 4:1716–1733, 2001.

[47] GNU. Dotgnu project. See http://dotgnu.info.

[48] I. Goryanin, T.C. Hodgman, and E. Selkov. Mathematical simulation and analysis of cellular metabolism and regulation. *Bioinformatics*, 15 (9):749–758, 1999.

[49] J. W. Haefner. *Modeling Biological Systems: Principles and applications*. Springer, 1996.

[50] C. M. Hoffmann and M. J. O'Donnell. Pattern matching in trees. *J. ACM*, 29(1):68–95, 1982.

[51] C. M. Hoffmann and M. J. O'Donnell. Programming with equations. *ACM Trans. Program. Lang. Syst.*, 4(1):83–112, 1982.

[52] J. Hughes. Why functional programming matters. pages 17–42, 1990.

[53] International ECMA. Ecma standard 334: Csharp language specification. See http://www.ecma-international.org/publications/standards/Ecma-334.htm.

[54] International ECMA. Ecma standard 335: Common language infrastructure. See http://www.ecma-international.org/publications/standards/Ecma-335.htm.

[55] S. James, P. Nilsson, G. James, S. Kjelleberg, and T. Fagerstroem. Luminescence control in the marine bacterium vibrio fischeri: An analysis of the dynamics of lux regulation. *Journal of Molecular Biology*, 296:1127–1137, 2000.

[56] P. Kilpeläinen. Tree matching problems with applications to structured text databases. November 1992.

[57] H. Kitano. *Foundations of System Biology*. MIT Press, 2001.

[58] H. Kitano. Computational system biology. *Nature*, 420:206–210, 2002.

[59] H. Kitano. A graphical notation fo biochemical networks. *Biosilico*, 1(5):169–176, 2003.

[60] J. W. Klop. Term rewriting systems. *Handbook of Logic in Computer Science*, 2:1–117, 1992.

[61] R. Knies. Making computer systems reveal biological secrets. See http://research.microsoft.com/displayArticle.aspx?id=1673, 2007.

[62] D.E. Knuth, J.R. Morris, and V.R. Pratt. Fast pattern matching in strings. *SIAM Journal on Computing*, 6(1):323–350, 1977.

[63] K. W. Kohn. Molecular interaction map of the mammalian cell cycle control and dna repair systems. *Molecular Biology of the Cell*, 10:2703–2734, 1999.

[64] K. W. Kohn. Molecular interaction maps as information organizers and simulation guides. *CHAOS*, 11(1):84–97, 2001.

[65] K. W. Kohn and M. I. Aladjem. Circuit diagrams for biological networks. *Molecular Systems Biology*, 2006. doi: 10.1038:msb4100044.

[66] K. W. Kohn, M. I. Aladjem, J. N. Weinstein, and Y. Pommier. Molecular interaction maps of bioregulatory networks: A general rubric for systems biology. *Molecular Biology of the Cell*, 17:1–13, 2006.

[67] S. R. Kosaraju. Efficient tree pattern matching. *IEEE - Foundations of Computer Science, 30th Annual Symposium on*, pages 178–183, 1989.

[68] L.Cardelli. Biological systems as complex system. *Contribution to FP7 Orientation paper on Complex Systems*, 2005.

[69] H. T. Lu and W. Yang. A simple tree pattern-matching algorithm. 2000.

[70] F. Luccio, A. M. Enriquez, P. O. Rieumont, and L. Pagli. Exact rooted subtree matching in sublinear time. Technical report, Universit di Pisa, Dipartimento di Informatica, 2001.

[71] F. Luccio, A. M. Enriquez, P. O. Rieumont, and L. Pagli. Bottom-up subtree isomorphism for unordered labeled trees. Technical report, Universit di Pisa, Dipartimento di Informatica, 2004.

[72] I. Marini, L. Bucchioni, P. Borella, A. Del Corso, and U. Mura. Sorbitol dehydrogenase from bovine lens: Purification and properties. *Archives of Biochemistry and Biophysics*, 370:383–391, 1997.

[73] H. Matsuno, A. Doi, M. Nagasaki, and S. Miyano. Hybrid petri net representation of gene regulatory network. *Pacific Symposium on Biocomputing*, pages 341–352, 2000.

[74] P. Mendes. Biochemistry by numbers: simulation of biochemical pathways with gepasi 3. *Trends Biochem. Sci.*, 22:361–363, 1997.

[75] T. C. Meng, S. S., and P. Dhar. Modeling and simulation of biological systems with stochasticity. *In silicio Biology ISB*, 4, 2004.

[76] Microsoft Corporation. .NET Framework.
See http://www.microsoft.com/net/.

[77] P. Milazzo. *Qualitative and Quantitative Formal Modeling of Biological Systems*. PhD thesis, Department of Computer Science, University of Pisa, Italy, April 2007.

[78] R. Milner. *Communicationg and mobile systems : the pi-calculus*. Cambridge University Press, 1999.

[79] M. Odersky, M. Sulzmann, and M. Wehr. Type inference with constrained types. *Theor. Pract. Object Syst.*, 5(1):35–55, 1999.

[80] E. M. Ozbudak, M. Thattai, I. Kurtser, A. D. Grossman, and A. van Oudenaarden. Regulation of noise in the expression of a single gene. *Nature Genetics*, 1:69–73, 2002.

[81] A. Phillips. The Stochastic Pi Machine (SPiM).
See http://research.microsoft.com/~aphillip/spim/.

[82] A. Phillips and L. Cardelli. A correct abstract machine for the stochastic pi-calculus. *BioConcur (Workshop on Concurrent Models in Molecular Biology)*, Electronic Notes in Theoretical Computer Science, 2004.

[83] R. Pickering. *Foundations of F#*. Apress, 2007.

[84] Mario Pineda-Krch. Marios Entangled Bank, 2007.
See http://pineda-krch.com/.

[85] C. Priami, A. Regev, E.Y. Shapiro, and W. Silverman. Application of a stochastic name-passing calculus to representation and simulation of molecular processes. *Information Processing Letters*, 80:25–31, 2001.

[86] G. Păun. *Membrane Computing. An Introduction.* Springer, 2002.

[87] A. Regev and E. Shapiro. Cells as computation. *Nature*, 419:343, 2002.

[88] A. Regev and E. Shapiro. The π–calculus as an abstraction for biomolecular systems. *Modelling in Molecular Biology*, Natural Computing Series, Springer:219–266, 2004.

[89] Davide Sangiorgi and David Walker. *The π-calculus: A Theory of Mobile Processes*. Cambridge University Press, 2001.

[90] D. Shasha and K. Zhang. Approximate tree pattern matching. pages 341–371, 1997.

[91] D. Stewart, H. Chaffey-Millar, G. Keller, Manuel M. T. Chakravarty, and C. Barner-Kowollik. Generative code specialisation for high-performance monte-carlo simulations. *SCKCB07*, 2007.
See http://www.cse.unsw.edu.au/~dons/polymer.html.

[92] S. H. Strogatz. *Non-Linear Dynamics and Chaos, with Applications to Physics, Biology, Chemistry and Engeneering.* Perseus books, 1994.

[93] D. Syme. ILX: Extending the .NET common IL for functional language interoperability. *Electronic Notes in Theoretical Computer Science*, 59(1), 2001.

[94] D. Syme. Leveraging .net meta-programming components from fsharp: Integrated queries and interoperable heterogeneous execution. In *ML Workshop (to be published)*, 2006.

[95] D. Syme, A Granicz, and A. Cisternino. *Expert F#*. Apress, 2007.

[96] D. Syme and J. Margetson. The F# website.
See http://research.microsoft.com/fsharp/.

[97] Don Syme. Introduction to f#. talk at microsoft research summer school 2006.
See http://research.microsoft.com/ero/phd/2006SummerSchool/DonSyme.ppt.

[98] Z. Szallasi, J. Stelling, and V. Periwal, editors. *System Modeling in Cellular Biology*. MIT Press, 2006.

[99] K. Takahashi. See http://www.e-cell.org/software/e-cell-system.

[100] M. Tomita, K. Hashimoto, K. Takahasi, T. Shimizu, Y. Matsuzaki, F. Miyoshi, K. Saito, S. Tanida, K. Yugi, J. Venter, and C. A. Hutchison. E-cell: software environment for whole-cell simulation. *Bioinformatics*, 15:72–84, 1999.

[101] G. Valiente. *Algorithms on Trees and Graphs*. Springer-Verlag Berlin, 2002.

[102] E. O. Voit. *Computational Analysis of Biochemical Systems: A Practical Guide for Biochemists and Molecular Biologists*. Cambridge University Press, Cambridge, UK, 2000.

[103] J. P. Ward, J. R. King, A. J. Koerber, J. M. Croft, R. E. Sockett, and P. Williams. Early development and quorum sensing in bacterial biofilms. *Journal of Mathematical Biology*, 47:23–55, 2003.

[104] D. Wilkinson. *Stochastic Modelling for Systems Biology.* Chapman & Hall/CRC, 2006.

[105] P. Wonga, S. Gladney, and J.D. Keasling. Mathematical model of the lac operon: Inducer exclusion, catabolite repression, and diauxic growth on glucose and lactose. *Biotechnology Progress*, 13:132–143, 1997.

[106] Ximian. Mono project.
See http://www.mono-project.com/.

[107] Young and Elcock. The kinetic montecarlo method. *Proceedings of the Physical Society*, 89:735, 1966.

[108] K. Zhang, R. Statman, and D. Sasha. On editing distance between unordered labeled trees. *Information Processing Letters*, 42:133–139, 1992.